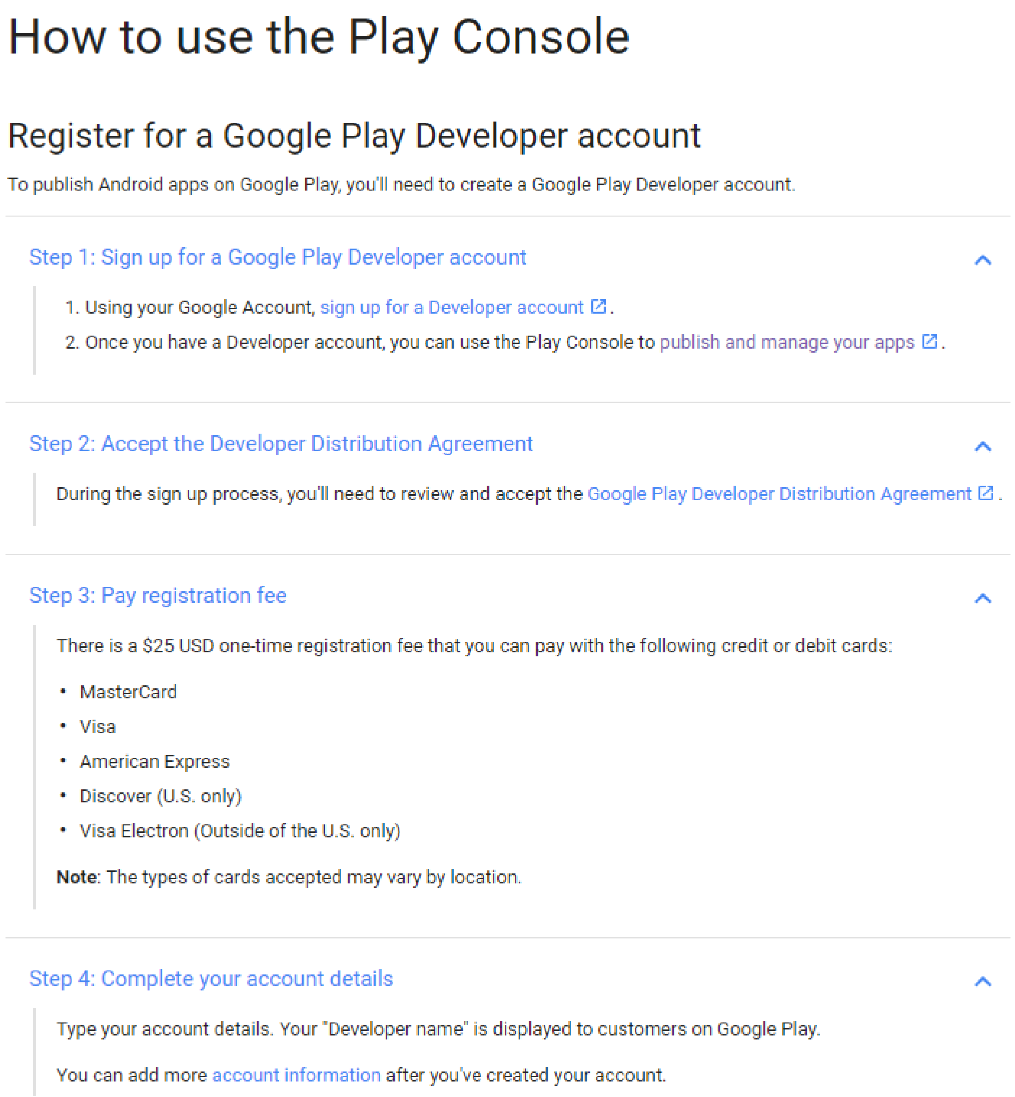
Exhibit 2b

Infringement Contentions – ASICS Digital's Android Mobile Apps

U.S. Patent No. 5,991,399

Claim	Analysis
1. A method of securely distributing data comprising:	ASICS Digital's mobile software application products and services including by way of example, but not limited to the following apps ("mobile applications", "mobile apps" or "Accused Products") that are specifically developed, used, sold, offered for sale, marketed, licensed and distributed by ASICS Digital to be downloaded onto Android mobile or tablet devices. • Runkeeper - GPS Track Run Walk (https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro), Last accessed on Mar 19, 2020 • ASICS Studio: Run, Lift & Get Fit (https://play.google.com/store/apps/details?id=com.fitnesskeeper.asicsstudio), Last accessed on Mar 19, 2020 ASICS Digital directly infringes and/or continues to knowingly induce Google to infringe this claim by intentionally developing, making, marketing, advertising, providing, sending, distributing and licensing its mobile applications software, documentation, materials, training or support and aiding, abetting, encouraging, promoting or inviting use thereof. To the extent any steps identified herein are performed by Google, such acts are attributable to ASICS Digital (i) because ASICS Digital works together with Google in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Google distributes and markets ASICS Digital's mobile apps under the direction and control of ASICS Digital, or acts as agent, or on behalf of ASICS Digital, in the building, marketing and distribution of ASICS Digital's mobile apps. Alternatively, any steps or acts performed by ASICS Digital, are attributable to Google, who conditions participation in and the receipt of a benefit, namely, the distribution of ASICS Digital's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Google in the building and upload of ASICS Digital's mobile apps, and ASICS Digital induces infringement by Google in the building, marketing and distribution of ASICS Digital's mobile apps. To the extent the preamble is limiting, ASICS Digital distributes data according to the method of claim 1 as set forth below. In order to build and send the mobile app securely to the Google servers, ASICS Digital practices the method of claim 1 as set forth below in order to securely distribute its mobile app to ASICS Digital's customers through the Google app store. As described below, ASICS Digital registers with the

Android Developer Console (https://play.google.com/apps/publish/, Last accessed on Mar 19, 2020) in order to securely upload ASICS Digital's apps onto the Google app store, Google Play, as evidenced by the "https" in the URL.



How to use the Play Console

Register for a Google Play Developer account

To publish Android apps on Google Play, you'll need to create a Google Play Developer account.

Step 1: Sign up for a Google Play Developer account

1. Using your Google Account, sign up for a Developer account.
2. Once you have a Developer account, you can use the Play Console to publish and manage your apps.

Step 2: Accept the Developer Distribution Agreement

During the sign up process, you'll need to review and accept the Google Play Developer Distribution Agreement.

Step 3: Pay registration fee

There is a $25 USD one-time registration fee that you can pay with the following credit or debit cards:

- MasterCard
- Visa
- American Express
- Discover (U.S. only)
- Visa Electron (Outside of the U.S. only)

Note: The types of cards accepted may vary by location.

Step 4: Complete your account details

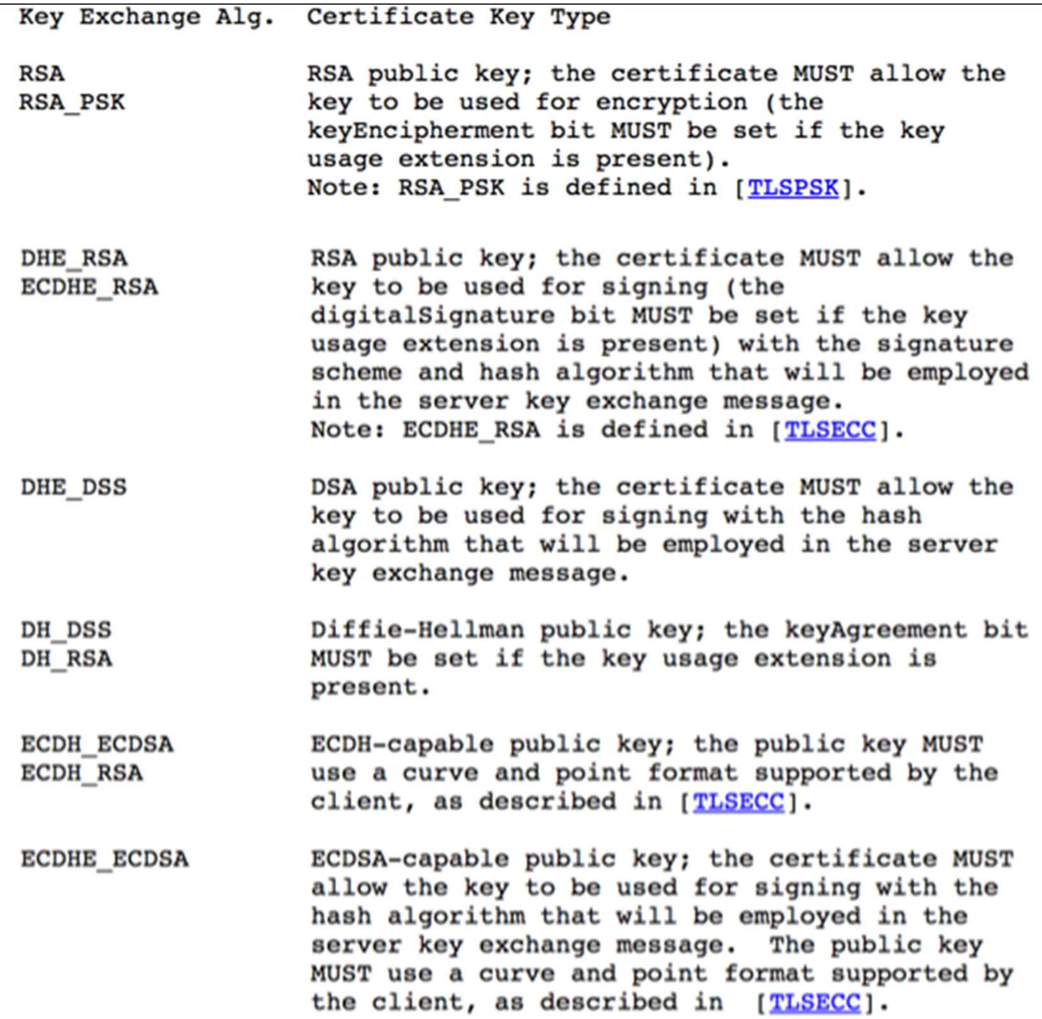
Type your account details. Your "Developer name" is displayed to customers on Google Play.

You can add more account information after you've created your account.

Source: https://support.google.com/googleplay/android-developer/answer/6112435, Last accessed on Mar 19, 2020

generating an asymmetric key pair having a public key and a private key;	The Court previously construed[1] "an asymmetric key pair having a public key and a private key" in claim 1 to mean "one or more asymmetric key pairs, one of the asymmetric key pairs having the claimed public key and claimed private key, the asymmetric keys of an asymmetric key pair being complementary by performing complementary functions, such as encrypting and decrypting data or creating and verifying signatures." Also relevant to this claim element is the Court's previous construction of "executable tamper resistant key module" / "executable tamper resistant code module" / "tamper resistant key module" to mean "software that is designed to work with other software, that is resistant to observation and modification, and that includes a key for secure communication." Also relevant to this claim element is the Court's rejection of limiting "including" to compiling. Upon information and belief, the method step of "generating an asymmetric key pair having a public key and a private key" is performed by Google and/or its agents – whose acts are attributable to ASICS Digital (i) because ASICS Digital works together with Google in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Google distributes and markets ASICS Digital's mobile apps under the direction and control of ASICS Digital, or acts as agent, or on behalf of ASICS Digital, in the building, marketing and distribution of ASICS Digital's mobile apps. Alternatively, to the extent any portion of this method step is performed by ASICS Digital, such acts are attributable to Google, who conditions participation in and the receipt of a benefit, namely, the distribution of ASICS Digital's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Google in the building and upload of ASICS Digital's mobile apps, and ASICS Digital induces infringement by Google in the building, marketing and distribution of ASICS Digital's mobile apps. ASICS Digital uploads their mobile apps to Google using a TLS connection – which begins with a TLS handshake. A TLS handshake is a mandatory procedure that allows ASICS Digital and Google to exchange cryptographic parameters, including a cipher suite and arrive at a shared master secret for encrypting communication including upload of ASICS Digital's mobile apps to Google servers. A TLS handshake begins with ASICS Digital sending a list of cipher suites supported by ASICS Digital to Google. These cipher suites specify at least one or more of the following key exchange algorithms:

[1] See Memorandum Opinion and Order, Document 104 signed by Judge Rodney Gilstrap on 7/21/2016 in re Plano Encryption Technologies, LLC v. American Bank of Texas (2:15-cv-01273).



```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].

DHE_RSA            RSA public key; the certificate MUST allow the
ECDHE_RSA          key to be used for signing (the
                   digitalSignature bit MUST be set if the key
                   usage extension is present) with the signature
                   scheme and hash algorithm that will be employed
                   in the server key exchange message.
                   Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS            DSA public key; the certificate MUST allow the
                   key to be used for signing with the hash
                   algorithm that will be employed in the server
                   key exchange message.

DH_DSS             Diffie-Hellman public key; the keyAgreement bit
DH_RSA             MUST be set if the key usage extension is
                   present.

ECDH_ECDSA         ECDH-capable public key; the public key MUST
ECDH_RSA           use a curve and point format supported by the
                   client, as described in [TLSECC].

ECDHE_ECDSA        ECDSA-capable public key; the certificate MUST
                   allow the key to be used for signing with the
                   hash algorithm that will be employed in the
                   server key exchange message.  The public key
                   MUST use a curve and point format supported by
                   the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

Each of these algorithms necessitates generating one or more asymmetric key pairs.

For **RSA and RSA _PSK**, a Google server generates an RSA public-private key pair. The RSA public key and the RSA private key are complementary, by performing complementary function encrypting and decrypting data.

	For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, Google server generates an RSA public-private key pair as well as a Diffie-Hellman ("DH") public-private key pair[2]. ASICS Digital also generates a second Diffie-Hellman public-private key pair. The RSA public key and the RSA private key are complementary by performing complementary functions, such as creating and verifying signatures. The Diffie-Hellman public key and the Diffie-Hellman private key are also complementary by performing complementary functions, such as encrypting and decrypting data. Specifically, as per the Diffie-Hellman key exchange algorithm, ASICS Digital uses Google's Diffie-Hellman public key combined with ASICS Digital's own Diffie-Hellman private key to compute a premaster secret and thereon a master secret[3] for encrypting data. Google in turn uses ASICS Digital's Diffie-Hellman public key combined with Google's own Diffie-Hellman private key to compute the same premaster secret and the master secret for decrypting encrypted data from ASICS Digital. Conversely, Google uses ASICS Digital's Diffie-Hellman public key combined with Google's own Diffie-Hellman private key to compute a premaster secret and thereon, a master secret[4] for encrypting data. ASICS Digital uses Google's Diffie-Hellman public key combined with ASICS Digital's own Diffie-Hellman private key to compute a premaster secret and thereon a master secret for decrypting encrypted data from Google. For **DHE_DSS and DH_DSS**, Google server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. ASICS Digital also generates a second Diffie-Hellman public-private key pair. The DSA public key and the DSA private key are complementary by performing complementary functions, such as creating and verifying signatures. The Diffie-Hellman public key and the Diffie-Hellman private key are also complementary by performing complementary functions, such as encrypting and decrypting data. Specifically, as per the Diffie-Hellman key exchange algorithm, ASICS Digital uses Google's Diffie-Hellman public key combined with ASICS Digital's own Diffie-Hellman private key to compute a premaster secret and thereon a master secret[5] for encrypting data. Google in turn uses ASICS Digital's Diffie-Hellman public key combined with Google's own Diffie-Hellman private key to compute the same premaster secret and the master secret for decrypting encrypted data from ASICS Digital. Conversely, Google uses ASICS Digital's Diffie-Hellman public key combined with Google's own Diffie-Hellman private key to compute a

[2] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN 1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020

[3] The master secret obtained from the premaster secret may be hashed according to a hashing algorithm also specified in the cipher suite in order to remove weak bits, as explained in https://tools.ietf.org/html/rfc5246, Sections 6.3, 7.4.9 and 8.1.

[4] The master secret obtained from the premaster secret may be hashed according to a hashing algorithm also specified in the cipher suite in order to remove weak bits, as explained in https://tools.ietf.org/html/rfc5246, Sections 6.3, 7.4.9 and 8.1.

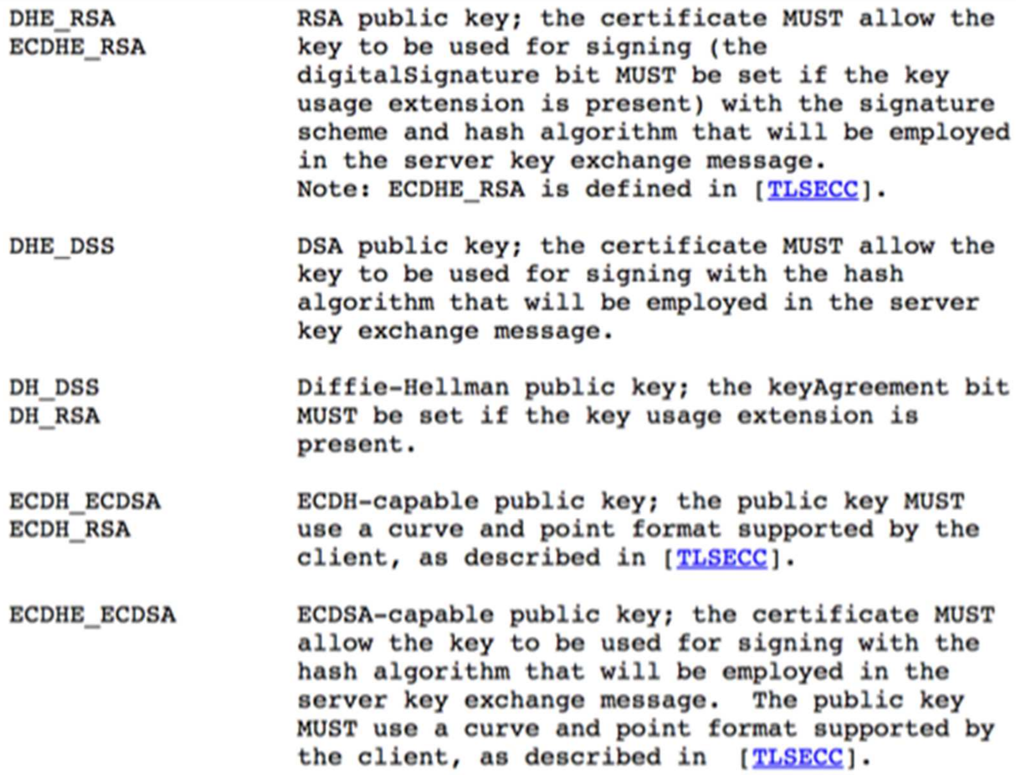
[5] The master secret obtained from the premaster secret may be hashed according to a hashing algorithm also specified in the cipher suite in order to remove weak bits, as explained in https://tools.ietf.org/html/rfc5246, Sections 6.3, 7.4.9 and 8.1.

	premaster secret and thereon, a master secret[6] for encrypting data. ASICS Digital uses Google's Diffie-Hellman public key combined with ASICS Digital's own Diffie-Hellman private key to compute a premaster secret and thereon a master secret for decrypting encrypted data from Google. For **ECDH_ECDSA and ECDHE_ECDSA**, a Google server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. ASICS Digital also generates a second Diffie-Hellman public-private key pair. The ECDSA public key and the ECDSA private key are complementary by performing complementary functions, such as creating and verifying signatures. The Diffie-Hellman public key and the Diffie-Hellman private key are also complementary by performing complementary functions, such as encrypting and decrypting data. Specifically, as per the Diffie-Hellman key exchange algorithm, ASICS Digital uses Google's Diffie-Hellman public key combined with ASICS Digital's own Diffie-Hellman private key to compute a premaster secret and thereon a master secret[7] for encrypting data. Google in turn uses ASICS Digital's Diffie-Hellman public key combined with Google's own Diffie-Hellman private key to compute the same premaster secret and the master secret for decrypting encrypted data from ASICS Digital. Conversely, Google uses ASICS Digital's Diffie-Hellman public key combined with Google's own Diffie-Hellman private key to compute a premaster secret and thereon, a master secret[8] for encrypting data. ASICS Digital uses Google's Diffie-Hellman public key combined with ASICS Digital's own Diffie-Hellman private key to compute a premaster secret and thereon a master secret for decrypting encrypted data from Google.

[6] The master secret obtained from the premaster secret may be hashed according to a hashing algorithm also specified in the cipher suite in order to remove weak bits, as explained in https://tools.ietf.org/html/rfc5246, Sections 6.3, 7.4.9 and 8.1.

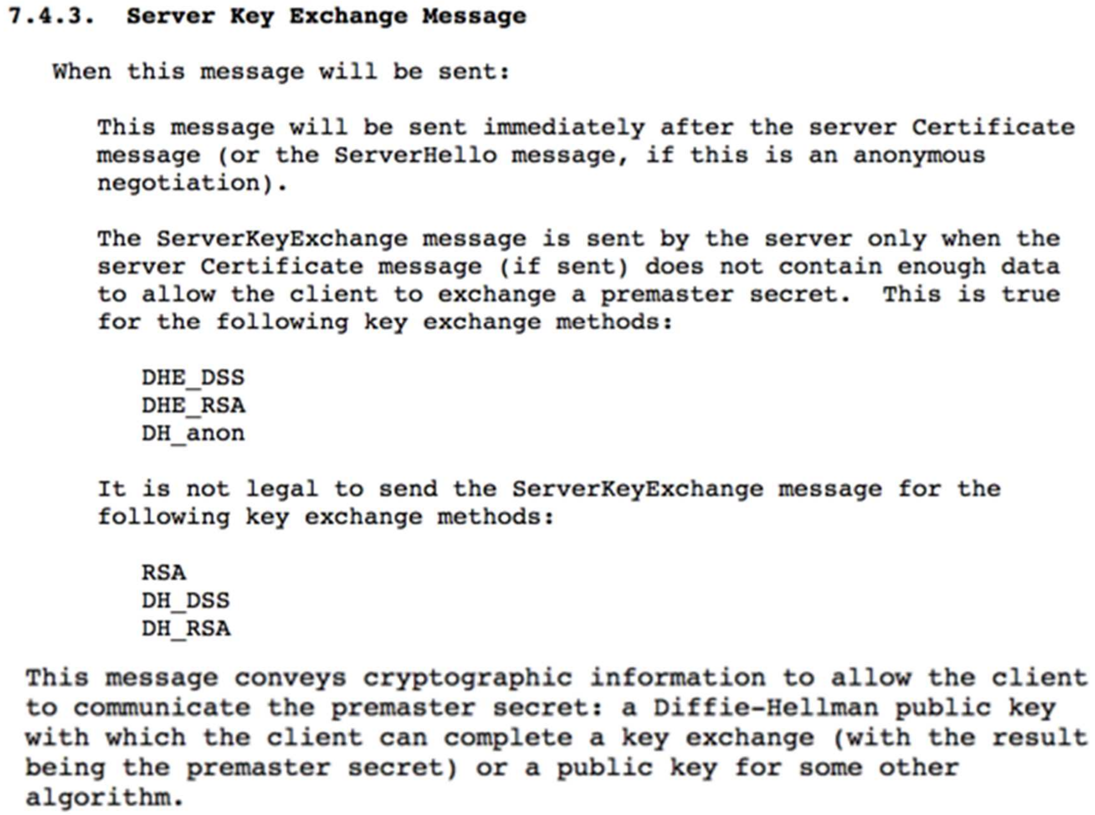
[7] The master secret obtained from the premaster secret may be hashed according to a hashing algorithm also specified in the cipher suite in order to remove weak bits, as explained in https://tools.ietf.org/html/rfc5246, Sections 6.3, 7.4.9 and 8.1.

[8] The master secret obtained from the premaster secret may be hashed according to a hashing algorithm also specified in the cipher suite in order to remove weak bits, as explained in https://tools.ietf.org/html/rfc5246, Sections 6.3, 7.4.9 and 8.1.



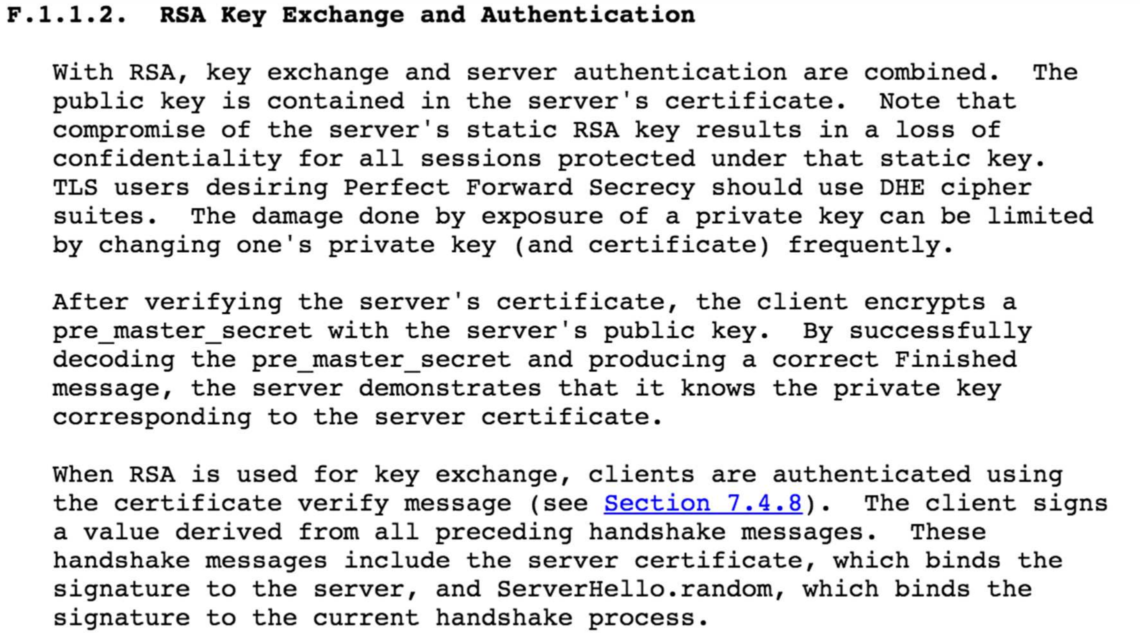
```
      DHE_RSA              RSA public key; the certificate MUST allow the
      ECDHE_RSA            key to be used for signing (the
                           digitalSignature bit MUST be set if the key
                           usage extension is present) with the signature
                           scheme and hash algorithm that will be employed
                           in the server key exchange message.
                           Note: ECDHE_RSA is defined in [TLSECC].

      DHE_DSS              DSA public key; the certificate MUST allow the
                           key to be used for signing with the hash
                           algorithm that will be employed in the server
                           key exchange message.

      DH_DSS               Diffie-Hellman public key; the keyAgreement bit
      DH_RSA               MUST be set if the key usage extension is
                           present.

      ECDH_ECDSA           ECDH-capable public key; the public key MUST
      ECDH_RSA             use a curve and point format supported by the
                           client, as described in [TLSECC].

      ECDHE_ECDSA          ECDSA-capable public key; the certificate MUST
                           allow the key to be used for signing with the
                           hash algorithm that will be employed in the
                           server key exchange message.  The public key
                           MUST use a curve and point format supported by
                           the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020



```
7.4.3.  Server Key Exchange Message

   When this message will be sent:

      This message will be sent immediately after the server Certificate
      message (or the ServerHello message, if this is an anonymous
      negotiation).

      The ServerKeyExchange message is sent by the server only when the
      server Certificate message (if sent) does not contain enough data
      to allow the client to exchange a premaster secret.  This is true
      for the following key exchange methods:

         DHE_DSS
         DHE_RSA
         DH_anon

      It is not legal to send the ServerKeyExchange message for the
      following key exchange methods:

         RSA
         DH_DSS
         DH_RSA

 This message conveys cryptographic information to allow the client
 to communicate the premaster secret: a Diffie-Hellman public key
 with which the client can complete a key exchange (with the result
 being the premaster secret) or a public key for some other
 algorithm.
```

Source: https://tools.ietf.org/html/rfc5246 at 50-51, Last accessed on Mar 19, 2020

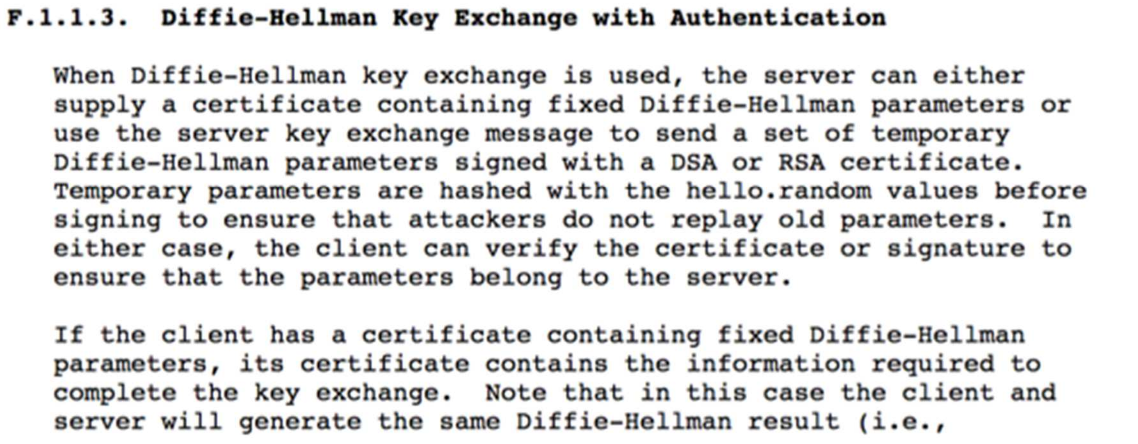


F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

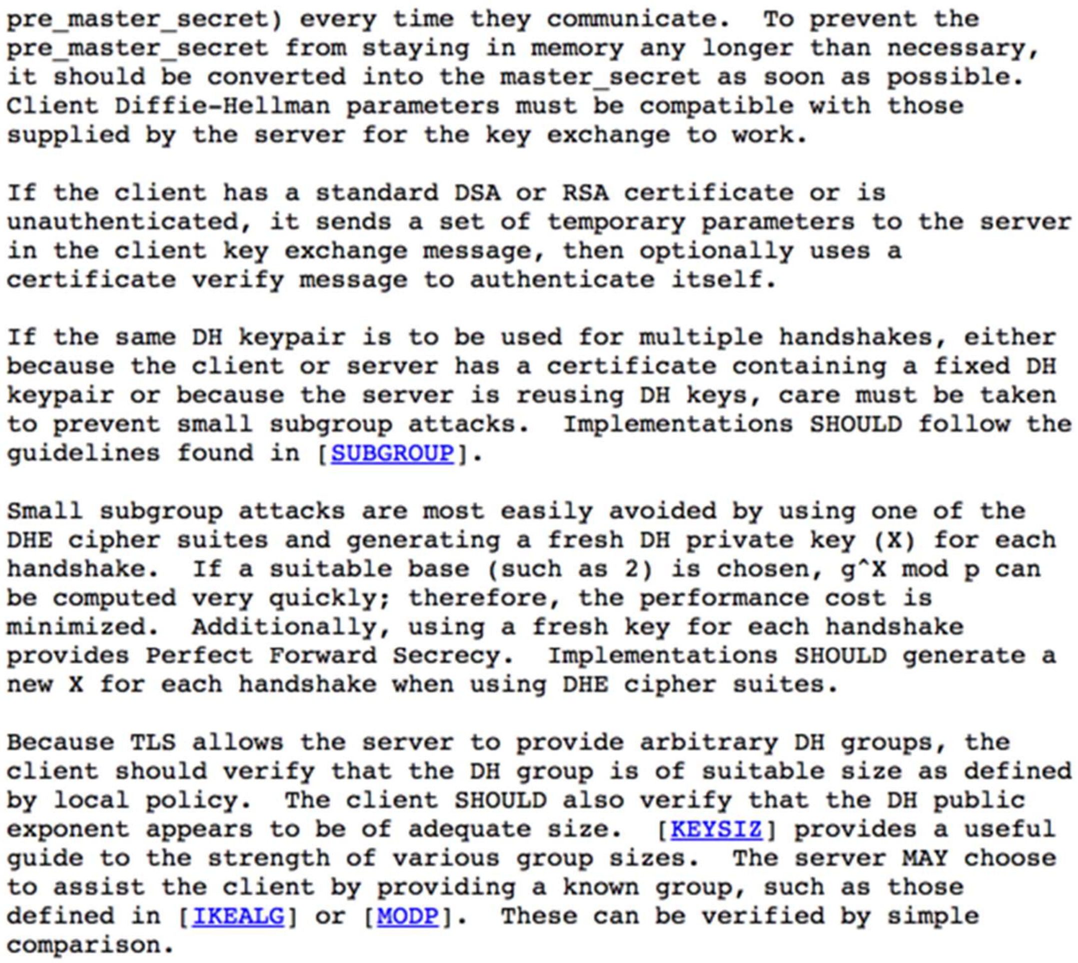
When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.



F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,



pre_master_secret) every time they communicate. To prevent the pre_master_secret from staying in memory any longer than necessary, it should be converted into the master_secret as soon as possible. Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work.

If the client has a standard DSA or RSA certificate or is unauthenticated, it sends a set of temporary parameters to the server in the client key exchange message, then optionally uses a certificate verify message to authenticate itself.

If the same DH keypair is to be used for multiple handshakes, either because the client or server has a certificate containing a fixed DH keypair or because the server is reusing DH keys, care must be taken to prevent small subgroup attacks. Implementations SHOULD follow the guidelines found in [SUBGROUP].

Small subgroup attacks are most easily avoided by using one of the DHE cipher suites and generating a fresh DH private key (X) for each handshake. If a suitable base (such as 2) is chosen, g^X mod p can be computed very quickly; therefore, the performance cost is minimized. Additionally, using a fresh key for each handshake provides Perfect Forward Secrecy. Implementations SHOULD generate a new X for each handshake when using DHE cipher suites.

Because TLS allows the server to provide arbitrary DH groups, the client should verify that the DH group is of suitable size as defined by local policy. The client SHOULD also verify that the DH public exponent appears to be of adequate size. [KEYSIZ] provides a useful guide to the strength of various group sizes. The server MAY choose to assist the client by providing a known group, such as those defined in [IKEALG] or [MODP]. These can be verified by simple comparison.

Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020

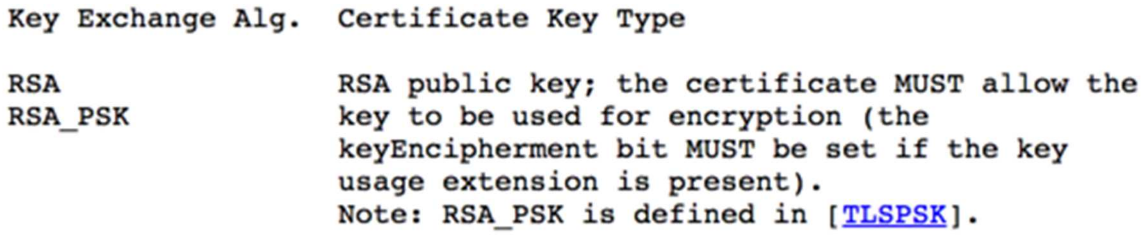
In addition, ASICS Digital and/or Google also generate additional asymmetric key pairs for code-signing the mobile app prior to upload and during TLS communications during the operation of the mobile app.

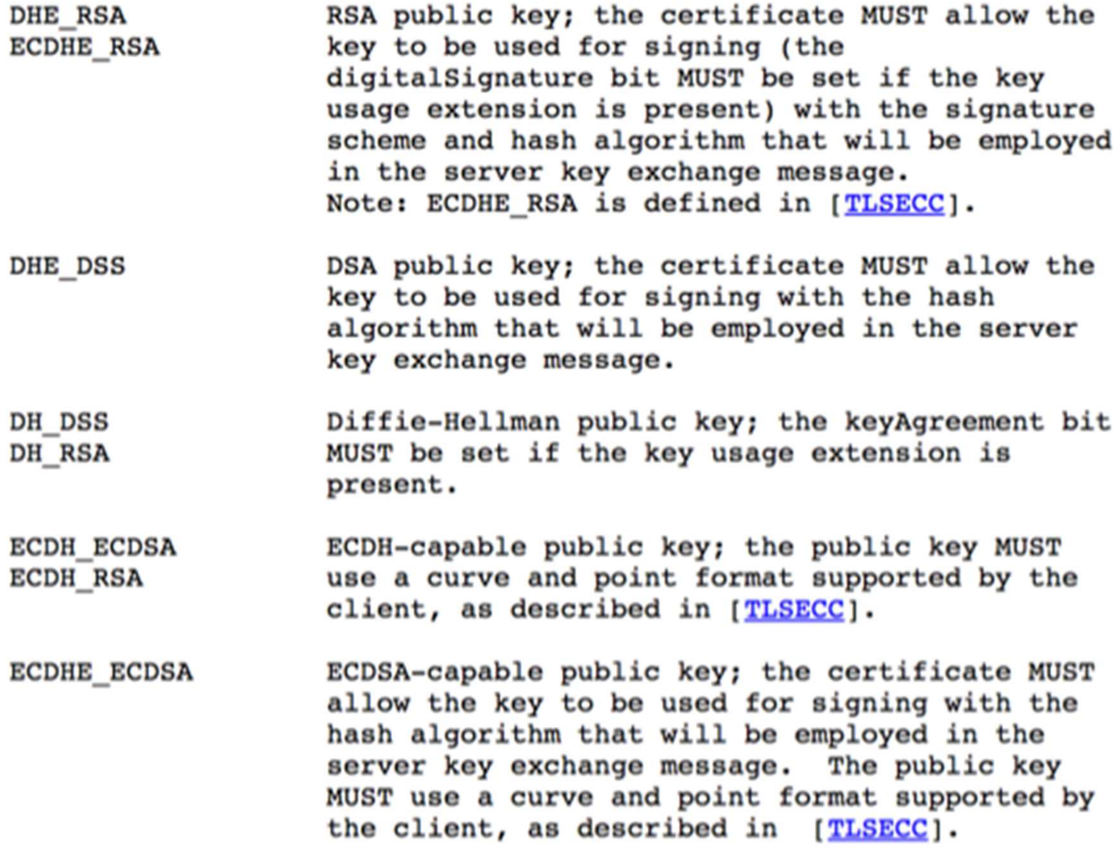
Thus, at least one asymmetric key pair is generated having the claimed public key and claimed private key in building the tamper resistant app so that the mobile app code and metadata can be sent securely to the Google servers by SSL/TLS. This asymmetric key pair (as with the asymmetric key

	pair used to digitally sign the app) is complementary as described below by performing complementary functions, such as encrypting and decrypting data and/or creating and verifying signatures. As described in greater detail below, the claimed public and private key are generated and used to securely upload the mobile app onto the Google servers by SSL/TLS when building the app, where the app includes the generated private key of the claimed asymmetric key pair and the encrypted predetermined data that has been encrypted with the generated public key of the claimed key pair.
encrypting predetermined data with the generated public key;	Upon information and belief, the method step of “encrypting predetermined data with the generated public key” is performed by ASICS Digital and/or its agents. To the extent any portion of the method step is performed by Google and/or its agents, such acts are attributable to ASICS Digital (i) because ASICS Digital works together with Google in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Google distributes and markets ASICS Digital’s mobile apps under the direction and control of ASICS Digital, or acts as agent, or on behalf of ASICS Digital, in the building, marketing and distribution of ASICS Digital’s mobile apps. When ASICS Digital uploads its mobile apps to Google, it connects to Google Play Developer Console using SSL/TLS protocol. Google and ASICS Digital perform a TLS handshake procedure which uses asymmetric key encryption and necessitates generation of at least one asymmetric key pair. Specifically, ASICS Digital negotiates with Google a cipher suite and the key exchange algorithm that will be used for the handshake. The cryptographic parameters of the session state are produced by the TLS Handshake Protocol, which operates on top of the TLS record layer. When a TLS client and server first start communicating, they agree on a protocol version, select cryptographic algorithms, optionally authenticate each other, and use public-key encryption techniques to generate shared secrets. The TLS Handshake Protocol involves the following steps: - Exchange hello messages to agree on algorithms, exchange random values, and check for session resumption. - Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret. - Exchange certificates and cryptographic information to allow the client and server to authenticate themselves.

	- Generate a master secret from the premaster secret and exchanged random values. - Provide security parameters to the record layer. - Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker. Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020 The actual key exchange uses up to four messages: the server Certificate, the ServerKeyExchange, the client Certificate, and the ClientKeyExchange. New key exchange methods can be created by specifying a format for these messages and by defining the use of the messages to allow the client and server to agree upon a shared secret. This secret MUST be quite long; currently defined key exchange methods exchange secrets that range from 46 bytes upwards. Following the hello messages, the server will send its certificate in a Certificate message if it is to be authenticated. Additionally, a ServerKeyExchange message may be sent, if it is required (e.g., if the server has no certificate, or if its certificate is for signing only). If the server is authenticated, it may request a certificate from the client, if that is appropriate to the cipher suite selected. Next, the server will send the ServerHelloDone message, indicating that the hello-message phase of the handshake is complete. The server will then wait for a client response. If the server has sent a CertificateRequest message, the client MUST send the Certificate message. The ClientKeyExchange message is now sent, and the content of that message will depend on the public key algorithm selected between the ClientHello and the ServerHello. If the client has sent a certificate with signing ability, a digitally-signed

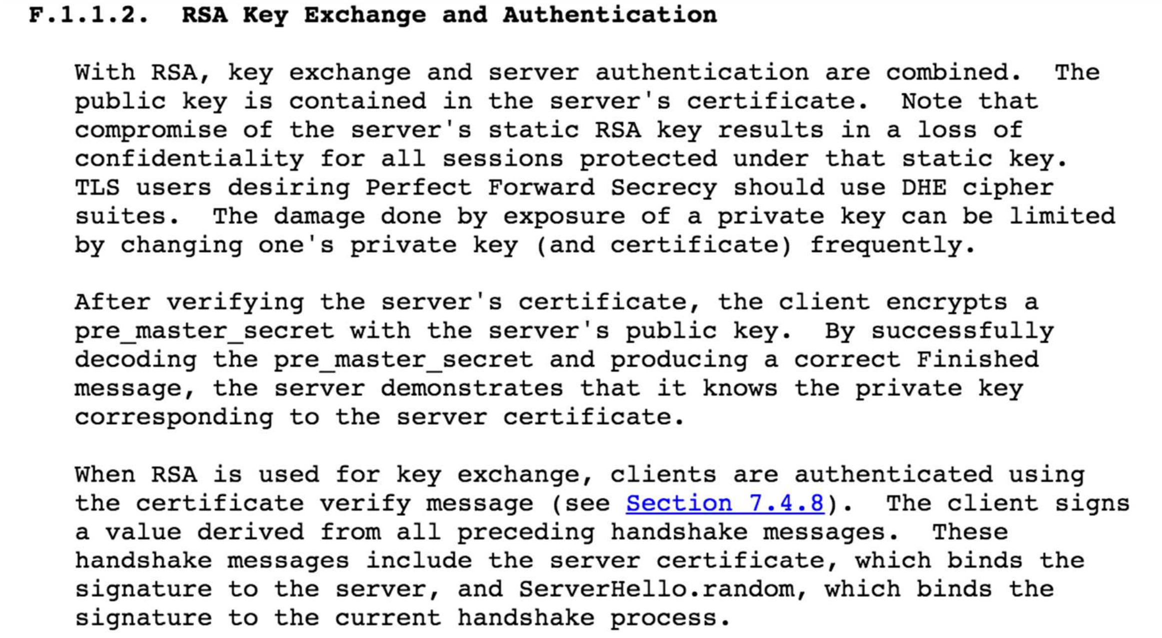
	CertificateVerify message is sent to explicitly verify possession of the private key in the certificate. Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020 **The TLS Handshaking Protocols** TLS has three subprotocols that are used to allow peers to agree upon security parameters for the record layer, to authenticate themselves, to instantiate negotiated security parameters, and to report error conditions to each other. The Handshake Protocol is responsible for negotiating a session, which consists of the following items: session identifier An arbitrary byte sequence chosen by the server to identify an active or resumable session state. peer certificate X509v3 [PKIX] certificate of the peer. This element of the state may be null. compression method The algorithm used to compress data prior to encryption. cipher spec Specifies the pseudorandom function (PRF) used to generate keying material, the bulk data encryption algorithm (such as null, AES, etc.) and the MAC algorithm (such as HMAC-SHA1). It also defines cryptographic attributes such as the mac_length. (See Appendix A.6 for formal definition.) master secret

	48-byte secret shared between the client and server. is resumable A flag indicating whether the session can be used to initiate new connections. These items are then used to create security parameters for use by the record layer when protecting application data. Many connections can be instantiated using the same session through the resumption feature of the TLS Handshake Protocol. Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020 As explained above, ASICS Digital and Google negotiate a key exchange algorithm from among the following key exchange algorithms:  Key Exchange Alg. Certificate Key Type RSA RSA public key; the certificate MUST allow the RSA_PSK key to be used for encryption (the keyEncipherment bit MUST be set if the key usage extension is present). Note: RSA_PSK is defined in [TLSPSK].



```
DHE_RSA            RSA public key; the certificate MUST allow the
ECDHE_RSA          key to be used for signing (the
                   digitalSignature bit MUST be set if the key
                   usage extension is present) with the signature
                   scheme and hash algorithm that will be employed
                   in the server key exchange message.
                   Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS            DSA public key; the certificate MUST allow the
                   key to be used for signing with the hash
                   algorithm that will be employed in the server
                   key exchange message.

DH_DSS             Diffie-Hellman public key; the keyAgreement bit
DH_RSA             MUST be set if the key usage extension is
                   present.

ECDH_ECDSA         ECDH-capable public key; the public key MUST
ECDH_RSA           use a curve and point format supported by the
                   client, as described in [TLSECC].

ECDHE_ECDSA        ECDSA-capable public key; the certificate MUST
                   allow the key to be used for signing with the
                   hash algorithm that will be employed in the
                   server key exchange message.  The public key
                   MUST use a curve and point format supported by
                   the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

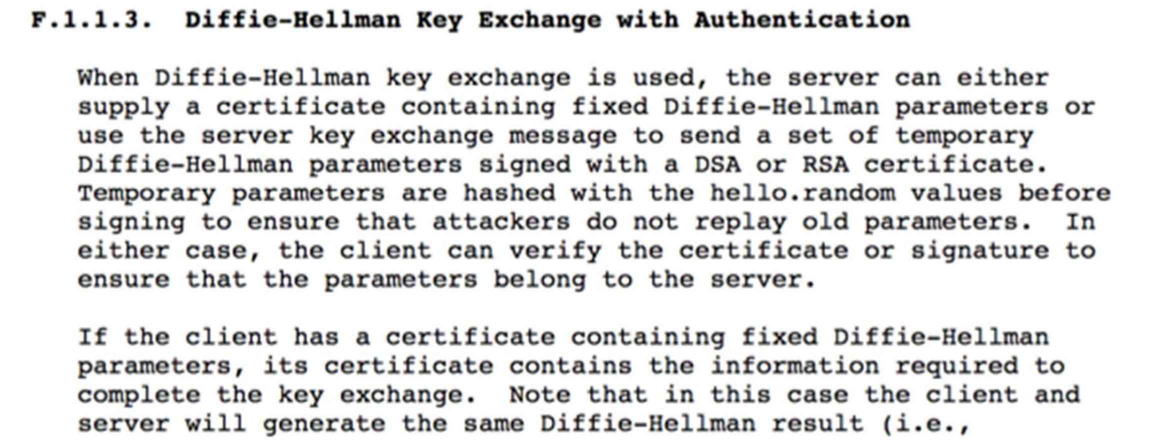


F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

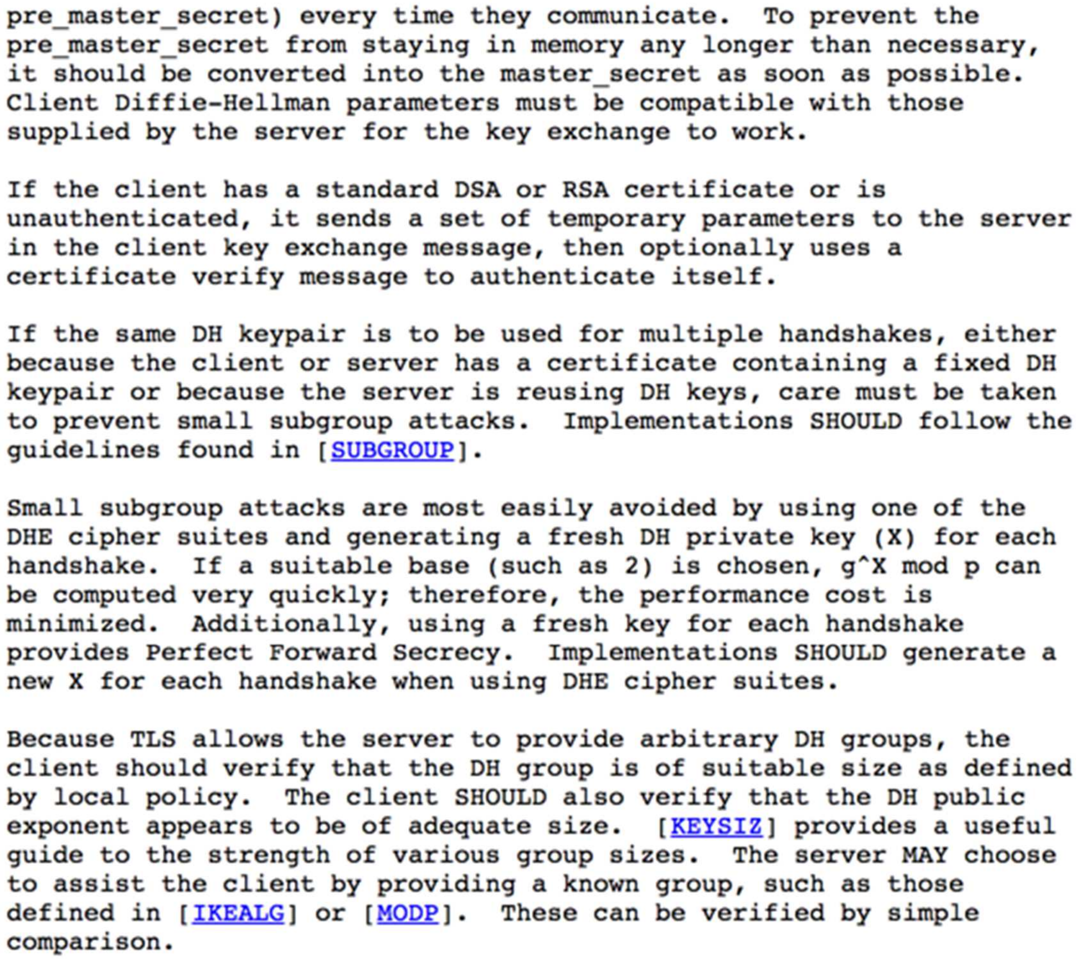
When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.



F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,



```
pre_master_secret) every time they communicate.  To prevent the
pre_master_secret from staying in memory any longer than necessary,
it should be converted into the master_secret as soon as possible.
Client Diffie-Hellman parameters must be compatible with those
supplied by the server for the key exchange to work.

If the client has a standard DSA or RSA certificate or is
unauthenticated, it sends a set of temporary parameters to the server
in the client key exchange message, then optionally uses a
certificate verify message to authenticate itself.

If the same DH keypair is to be used for multiple handshakes, either
because the client or server has a certificate containing a fixed DH
keypair or because the server is reusing DH keys, care must be taken
to prevent small subgroup attacks.  Implementations SHOULD follow the
guidelines found in [SUBGROUP].

Small subgroup attacks are most easily avoided by using one of the
DHE cipher suites and generating a fresh DH private key (X) for each
handshake.  If a suitable base (such as 2) is chosen, g^X mod p can
be computed very quickly; therefore, the performance cost is
minimized.  Additionally, using a fresh key for each handshake
provides Perfect Forward Secrecy.  Implementations SHOULD generate a
new X for each handshake when using DHE cipher suites.

Because TLS allows the server to provide arbitrary DH groups, the
client should verify that the DH group is of suitable size as defined
by local policy.  The client SHOULD also verify that the DH public
exponent appears to be of adequate size.  [KEYSIZ] provides a useful
guide to the strength of various group sizes.  The server MAY choose
to assist the client by providing a known group, such as those
defined in [IKEALG] or [MODP].  These can be verified by simple
comparison.
```

Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020

For each of the above key exchange algorithms, Google and/or ASICS Digital encrypts predetermined data using the generated public key(s).

For **RSA and RSA _PSK**, ASICS Digital encrypts a random premaster secret with Google's RSA public key and sends the encrypted premaster secret to Google. Google decrypts the premaster secret with its matched RSA private key. ASICS Digital and Google both use the premaster secret to compute a master secret which is then used by both ASICS Digital and Google to encrypt all subsequent communications between ASICS Digital and Google.

	Therefore, when any of **RSA and RSA _PSK** algorithms are chosen during a TLS handshake, ASICS Digital encrypts predetermined data (i.e. the premaster secret) with the generated public key (i.e. Google's RSA public key). For the other key exchange algorithms, namely Diffie-Hellman based algorithms such as **DHE_RSA, ECDHE_RSA, DH_RSA, DHE_DSS, DH_DSS, ECDH_RSA**, **ECDH_ECDSA and ECDHE_ECDSA,** ASICS Digital sends its Diffie-Hellman public key[9] to Google while Google sends its Diffie-Hellman public key to ASICS Digital. ASICS Digital then uses Google's Diffie-Hellman public key combined with ASICS Digital's own Diffie-Hellman private key to compute a premaster secret. Google in turn uses ASICS Digital's Diffie-Hellman public key combined with Google's own Diffie-Hellman private key to compute the same premaster secret. ASICS Digital and Google both use the premaster secret to compute a master secret[10] which is then used by both ASICS Digital and Google to encrypt all subsequent communications between ASICS Digital and Google. Therefore, when any of the Diffie-Hellman based key exchange algorithms are chosen during a TLS handshake, ASICS Digital encrypts predetermined data, i.e. all communication with Google subsequent to the TLS handshake, including at least the executable compiled code related to its mobile apps and information such as name, category, screenshots and description related to ASICS Digital's mobile apps, with the generated public key, i.e. Google's Diffie-Hellman public key, since the master secret used to encrypt the predetermined data is a combination of Google's Diffie-Hellman public key and ASICS Digital's Diffie-Hellman private key[11]. Similarly, when any of the Diffie-Hellman based key exchange algorithms are chosen during a TLS handshake, Google encrypts predetermined data, i.e. all communication with Google subsequent to the TLS handshake, including at least textual and graphic data and software relating to Google Play Developer Console website and forms that ASICS Digital uses for uploading its mobile app to Google Play Store, with the generated public key, i.e. ASICS Digital's Diffie-Hellman public key, since the master secret used to encrypt the predetermined data is a combination of ASICS Digital's Diffie-Hellman public key and Google's Diffie-Hellman private key[12].
and building an executable tamper resistant key module identified for a selected program, the executable	Relevant to this claim element is the Court's previous construction[13] of "executable tamper resistant key module" / "executable tamper resistant code module" / "tamper resistant key module" to mean "software that is designed to work with other software, that is resistant to observation and

[9] For Diffie-Hellman (DH) based algorithms such as DH_RSA, DHE_RSA, ECDH_RSA, ECDHE_RSA, DH_DSS, DHE_DSS, ECDH_ECDSA and ECDHE_ECDSA, Google calculates a hash of the message containing their Diffie-Hellman public key and encrypts the hash with their RSA/DSA/ECDSA private key (i.e. signing the message). Google then sends that RSA/DSA/ECDSA public key to ASICS Digital in a digital certificate so that ASICS Digital can authenticate the Google server by decrypting the hash using Google's public key and matching the decryption result to a hash of the received message as calculated by ASICS Digital itself. If the two values match, ASICS Digital knows that the message originated from Google and not from a malicious third party.

[10] The master secret obtained from the premaster secret may be hashed according to a hashing algorithm also specified in the cipher suite in order to remove weak bits, as explained in https://tools.ietf.org/html/rfc5246, Sections 6.3, 7.4.9 and 8.1.

[11] See, for example, https://tools.ietf.org/html/rfc2631, Page 2, Last accessed on Mar 19, 2020

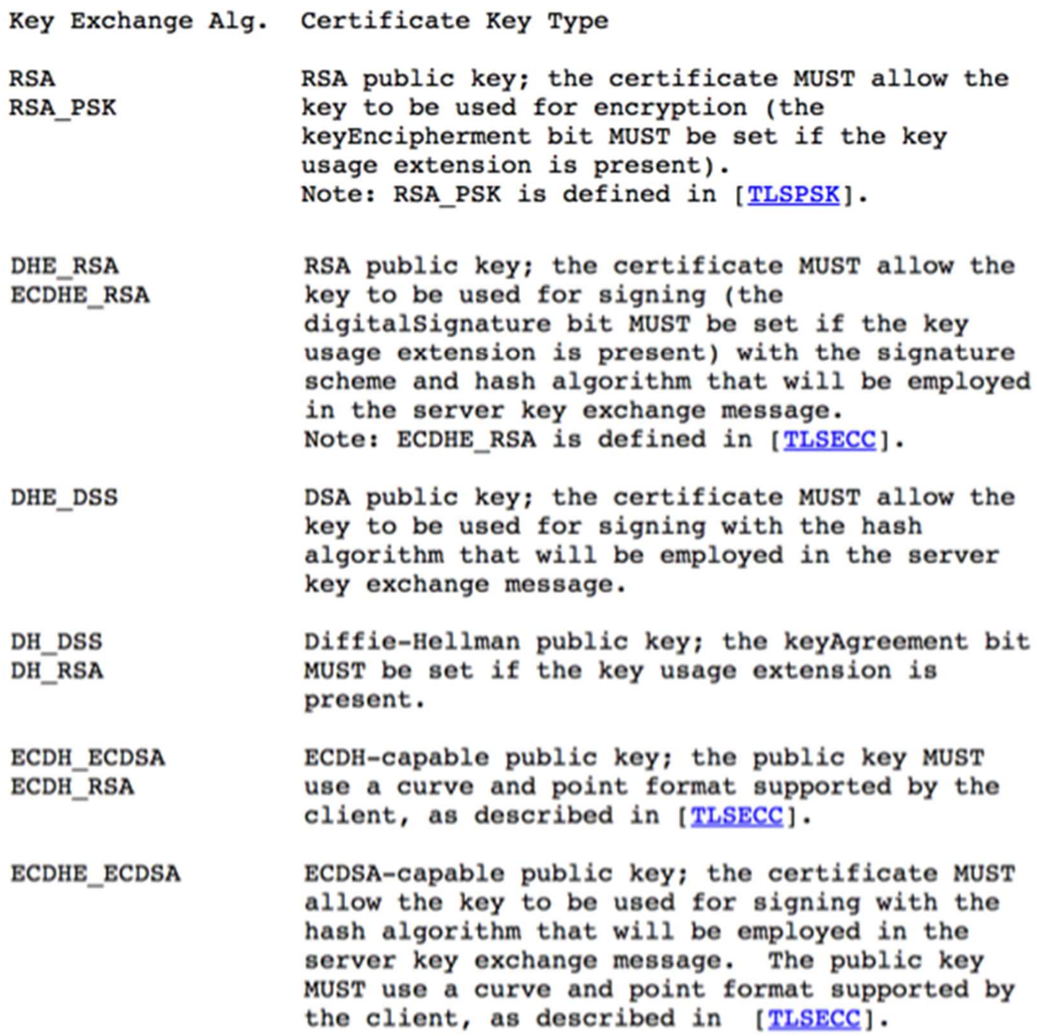
[12] Id.

[13] See Memorandum Opinion and Order, Document 104 signed by Judge Rodney Gilstrap on 7/21/2016 in re Plano Encryption Technologies, LLC v. American Bank of Texas (2:15-cv-01273).

tamper resistant key module including the generated private key and the encrypted predetermined data.	modification, and that includes a key for secure communication." Also relevant to this claim element is the Court's previous rejection of limiting "including" to compiling[14]. The method step of "building an executable tamper resistant key module identified for a selected program, the executable tamper resistant key module including the generated private key and the encrypted predetermined data" is performed by ASICS Digital and/or its agents. To the extent any portion of the method step is performed by Google and/or its agents, such acts are attributable to ASICS Digital (i) because ASICS Digital works together with Google in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Google distributes and markets ASICS Digital's mobile apps under the direction and control of ASICS Digital, or acts as agent, or on behalf of ASICS Digital, in the building, marketing and distribution of ASICS Digital's mobile apps. Building of an "executable tamper resistant code module" (that is, the mobile app) requires the inclusion of at least the following different asymmetric key pairs: (1) An asymmetric key pair must be included in order to send the mobile app from ASICS Digital to Google securely by SSL/TLS; and (2) An asymmetric key pair must be included by ASICS Digital in order to digitally sign the mobile app with a private asymmetric key and to verify the mobile app has not been changed with the public key for Android compatible mobile apps. The asymmetric key pair that is included to digitally sign the mobile app is different from and in addition to the claimed asymmetric key pair used to securely upload the mobile app to the Google servers for distribution on the Google Play Store. Thus, at least one asymmetric key pair is included having the claimed public key and claimed private key in building the tamper resistant app so that the mobile app code can be sent uploaded to the Google servers using SSL/TLS protocol. This asymmetric key pair(s) (as with the asymmetric key pair used to digitally sign the app) is complementary as described below by performing complementary functions, such as encrypting and decrypting data and/or creating and verifying signatures. As described in greater detail below, the claimed public and private key are generated and used to securely upload the mobile app onto the Google servers by SSL/TLS when building the app, where the app includes the generated private key of the claimed asymmetric key pair(s) and the encrypted predetermined data. ASICS Digital uploads their mobile apps to Google using a TLS connection – which begins with a TLS handshake. A TLS handshake is a mandatory procedure that allows ASICS Digital and Google to exchange cryptographic parameters, including a cipher suite and arrive at a shared master secret for encrypting communication including upload of ASICS Digital's mobile apps to Google servers.

[14] Also relevant is the Court's construction of "an asymmetric key pair having a public key and a private key" in claims 1, 9 and 10 to mean "one or more asymmetric key pairs, one of the asymmetric key pairs having the claimed public key and claimed private key, the asymmetric keys of an asymmetric key pair being complementary by performing complementary functions, such as encrypting and decrypting data or creating and verifying signatures."

A TLS handshake begins with ASICS Digital sending a list of cipher suites supported by ASICS Digital to Google. These cipher suites specify at least one or more of the following key exchange algorithms:



```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].

DHE_RSA            RSA public key; the certificate MUST allow the
ECDHE_RSA          key to be used for signing (the
                   digitalSignature bit MUST be set if the key
                   usage extension is present) with the signature
                   scheme and hash algorithm that will be employed
                   in the server key exchange message.
                   Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS            DSA public key; the certificate MUST allow the
                   key to be used for signing with the hash
                   algorithm that will be employed in the server
                   key exchange message.

DH_DSS             Diffie-Hellman public key; the keyAgreement bit
DH_RSA             MUST be set if the key usage extension is
                   present.

ECDH_ECDSA         ECDH-capable public key; the public key MUST
ECDH_RSA           use a curve and point format supported by the
                   client, as described in [TLSECC].

ECDHE_ECDSA        ECDSA-capable public key; the certificate MUST
                   allow the key to be used for signing with the
                   hash algorithm that will be employed in the
                   server key exchange message.  The public key
                   MUST use a curve and point format supported by
                   the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

	For each of the key exchange algorithms, ASICS Digital and/or Google build an executable tamper resistant key module that includes the generated private key and the encrypted predetermined data. For **RSA and RSA _PSK**, the executable tamper resistant key module includes encrypted predetermined data (i.e. the encrypted premaster secret) as it would be impossible for ASICS Digital to send its mobile app to Google using SSL/TLS without encrypting a premaster secret and sending it to Google. The executable tamper resistant key module also includes: 1. Google's RSA private key corresponding to Google's RSA public key used by ASICS Digital to encrypt the premaster secret. 2. ASICS Digital's private key used to code-sign the mobile app. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, the executable tamper resistant key module includes encrypted predetermined data (i.e. all communication with Google subsequent to the TLS handshake, including at least the executable compiled code related to its mobile apps and information such as name, category, screenshots and description related to ASICS Digital's mobile apps). The executable tamper resistant key module also includes: 1. Google's Diffie-Hellman private key corresponding to Google's Diffie-Hellman public key used by ASICS Digital to compute the shared master secret. 2. ASICS Digital's Diffie-Hellman private key used to compute the shared master secret. 3. Google's RSA private key used to sign the message containing Google's Diffie-Hellman public key. 4. ASICS Digital's private key used to code-sign the mobile app. For **DHE_DSS and DH_DSS**, the executable tamper resistant key module includes encrypted predetermined data (i.e. all communication with Google subsequent to the TLS handshake, including at least the executable compiled code related to its mobile apps and information such as name, category, screenshots and description related to ASICS Digital's mobile apps). The executable tamper resistant key module also includes: 1. Google's Diffie-Hellman private key corresponding to Google's Diffie-Hellman public key used by ASICS Digital to compute the shared master secret. 2. ASICS Digital's Diffie-Hellman private key used to compute the shared master secret. 3. Google's DSA private key used to sign the message containing Google's Diffie-Hellman public key. 4. ASICS Digital's private key used to code-sign the mobile app. For **ECDH_ECDSA and ECDHE_ECDSA**, the executable tamper resistant key module includes encrypted predetermined data (i.e. all communication with Google subsequent to the TLS handshake, including at least the executable compiled code related to its mobile apps and information such as name, category, screenshots and description related to ASICS Digital's mobile apps). The executable tamper resistant key module also includes: 1. Google's Diffie-Hellman private key corresponding to Google's Diffie-Hellman public key used by ASICS Digital to compute the shared master secret. 2. ASICS Digital's Diffie-Hellman private key used to compute the shared master secret.

	3. Google's ECDSA private key used to sign the message containing Google's Diffie-Hellman public key. 4. ASICS Digital's private key used to code-sign the mobile app. ASICS Digital builds an executable tamper resistant key module identified for a selected program resident on a remote system. Specifically, ASICS Digital builds a mobile app, which is an executable tamper resistant key module, as explained in more detail below. This mobile app is identified for a selected program resident on a remote system, namely the Android operating system on a remote mobile device. The mobile app comprises an executable tamper resistant key module that is identified for the Android program and includes the claimed private key described above and the encrypted predetermined data encrypted with the claimed public key also described above in building the mobile app so that it can be made available for download from Google servers onto devices compatible with Android operating system for use by customers of ASICS Digital. An asymmetric key pair is used not only to upload the binary code files for the mobile app, but an entire application package, including all of the metadata for the app, such as title, screenshots, and other resources or information such as application type, category, price, *etc*. which are included during the upload process so that the mobile app can be identified by potential users for download[15]. ASICS Digital's mobile app is each an executable tamper resistant key module because it is designed to work with other software, namely the Android operating system as well as other applications or programs installed on a user's mobile device; because the mobile app is resistant to observation and modification, as explained below; and because in building ASICS Digital mobile apps on the Google platform, ASICS Digital's apps include at least the claimed generated private key and the encrypted predetermined data including by way of example, the pre-master secret encrypted with the claimed generated public key when the mobile app is securely uploaded onto the Google servers as described above. The tamper resistant key module includes several keys "used for secure communications" per the Court's previous construction including at least the following: For **RSA and RSA _PSK**: 1. Google's RSA private key corresponding to Google's RSA public key used by ASICS Digital to encrypt the premaster secret. 2. ASICS Digital's private key used to code-sign the mobile app. 3. Symmetric key used for uploading the mobile app to Google subsequent to the TLS handshake. 4. Asymmetric keys and symmetric keys used for TLS communications during operation of the app. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**: 1. Google's Diffie-Hellman private key corresponding to Google's Diffie-Hellman public key used by ASICS Digital to compute the shared master secret. 2. ASICS Digital's Diffie-Hellman private key used to compute the shared master secret.

[15] *See, e.g.,* https://developer.android.com/studio/publish/index.html, Last accessed on Mar 19, 2020

	3. Google's RSA private key used to sign the message containing Google's Diffie-Hellman public key. 4. ASICS Digital's private key used to code-sign the mobile app. 5. Shared master secret key used for uploading the mobile app to Google subsequent to the TLS handshake. 6. Asymmetric keys and symmetric keys used for TLS communications during operation of the app. For **DHE_DSS and DH_DSS**: 1. Google's Diffie-Hellman private key corresponding to Google's Diffie-Hellman public key used by ASICS Digital to compute the shared master secret. 2. ASICS Digital's Diffie-Hellman private key used to compute the shared master secret. 3. Google's DSA private key used to sign the message containing Google's Diffie-Hellman public key. 4. ASICS Digital's private key used to code-sign the mobile app. 5. Shared master secret key used for uploading the mobile app to Google subsequent to the TLS handshake. 6. Asymmetric keys and symmetric keys used for TLS communications during operation of the app. For **ECDH_ECDSA and ECDHE_ECDSA**: 1. Google's Diffie-Hellman private key corresponding to Google's Diffie-Hellman public key used by ASICS Digital to compute the shared master secret. 2. ASICS Digital's Diffie-Hellman private key used to compute the shared master secret. 3. Google's ECDSA private key used to sign the message containing Google's Diffie-Hellman public key. 4. ASICS Digital's private key used to code-sign the mobile app. 5. Shared master secret key used for uploading the mobile app to Google subsequent to the TLS handshake. 6. Asymmetric keys and symmetric keys used for TLS communications during operation of the app. ASICS Digital's mobile apps are tamper resistant, resistant to observation and modification, as follows: **1. Resistant to Observation Because App Source Code Is Compiled Before Upload** ASICS Digital mobile apps are resistant to observation, at least in part, since ASICS Digital compiles its mobile app source code before submitting the app to Google – and uploads the binary output of the compilation process rather than the source code itself[16].

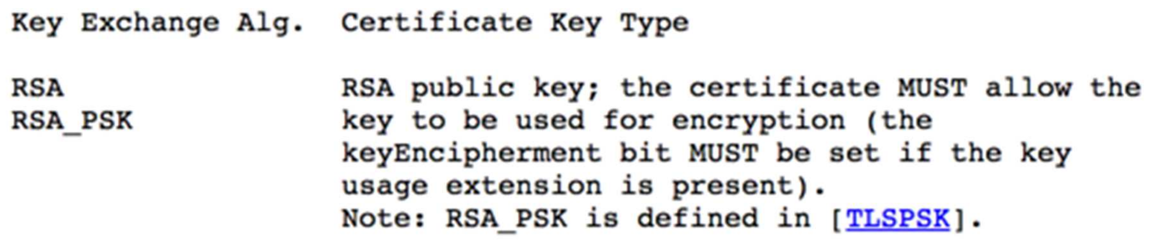
[16] *See, e.g.,* https://developer.android.com/studio/build/index.html, Last accessed on Mar 19, 2020

See, Android Studio Users Guide, “Prepare for Release” stating “To release your application to users you need to create a release-ready package that users can install and run on their Android-powered devices. The release-ready package contains the same components as the debug APK file — compiled source code, resources, manifest file, and so on — and it is built using the same build tools. However, unlike the debug APK file, the release-ready APK file is signed with your own certificate and it is optimized with the zipalign tool.” https://developer.android.com/studio/publish/preparing.html, Last accessed on Mar 19, 2020

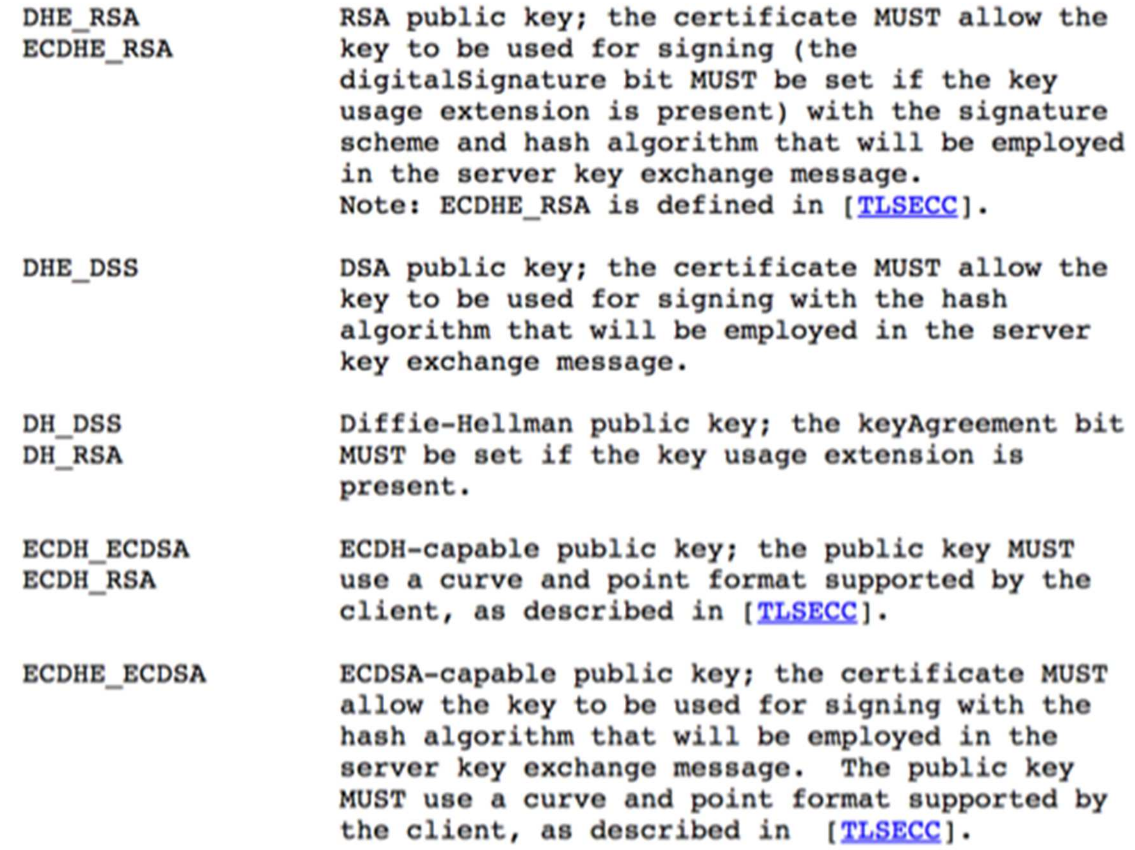
2. Resistant to Observation Because Upload To Google Is Over SSL/TLS

ASICS Digital’s mobile apps are made further resistant to observation, at least in part, because the mobile app is securely sent by SSL/TLS to Google as part of the building process. Android Developer Console (https://play.google.com/apps/publish/, Last accessed on Mar 19, 2020) establishes SSL/TLS communications when uploading ASICS Digital’s apps, as evidenced by the “https” in the URL. Sending the mobile app code by SSL/TLS is necessary to keep the code from being observed in transit from the code developer to Google.

The secure upload process starts with a TLS handshake procedure which uses asymmetric key encryption and necessitates generation of at least one asymmetric key pair. Specifically, ASICS Digital negotiates with Google the cipher suite and the key exchange algorithm that will be used for the handshake.

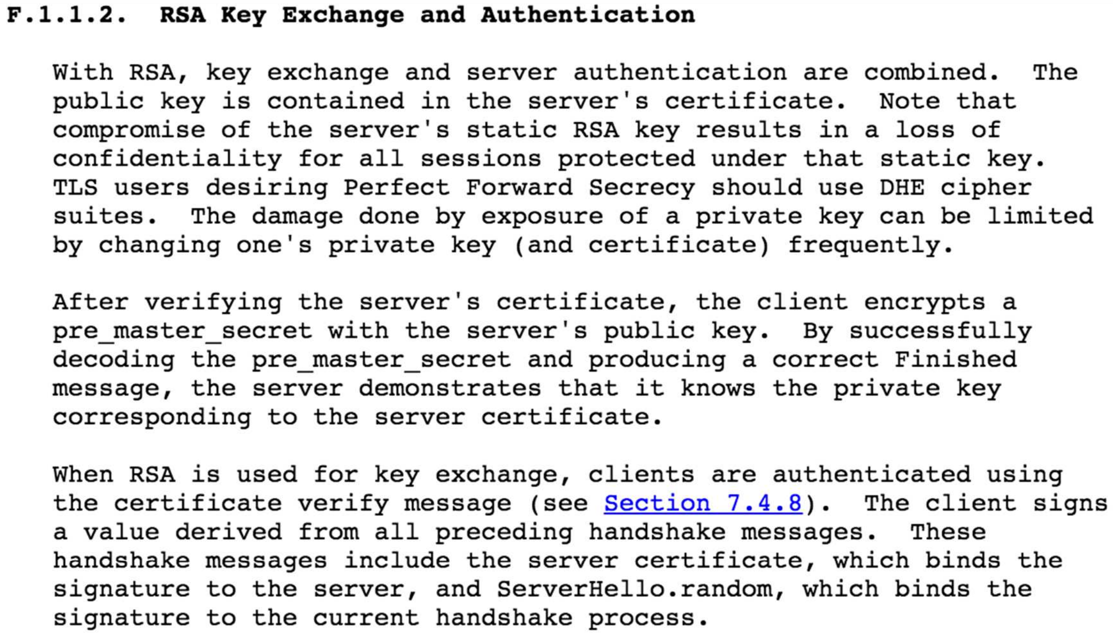


```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].
```



```
DHE_RSA            RSA public key; the certificate MUST allow the
ECDHE_RSA          key to be used for signing (the
                   digitalSignature bit MUST be set if the key
                   usage extension is present) with the signature
                   scheme and hash algorithm that will be employed
                   in the server key exchange message.
                   Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS            DSA public key; the certificate MUST allow the
                   key to be used for signing with the hash
                   algorithm that will be employed in the server
                   key exchange message.

DH_DSS             Diffie-Hellman public key; the keyAgreement bit
DH_RSA             MUST be set if the key usage extension is
                   present.

ECDH_ECDSA         ECDH-capable public key; the public key MUST
ECDH_RSA           use a curve and point format supported by the
                   client, as described in [TLSECC].

ECDHE_ECDSA        ECDSA-capable public key; the certificate MUST
                   allow the key to be used for signing with the
                   hash algorithm that will be employed in the
                   server key exchange message.  The public key
                   MUST use a curve and point format supported by
                   the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

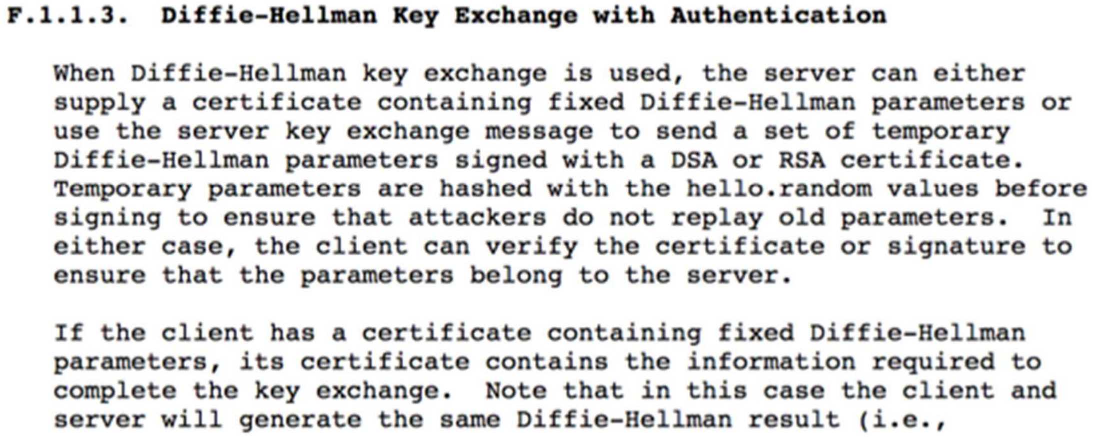


F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

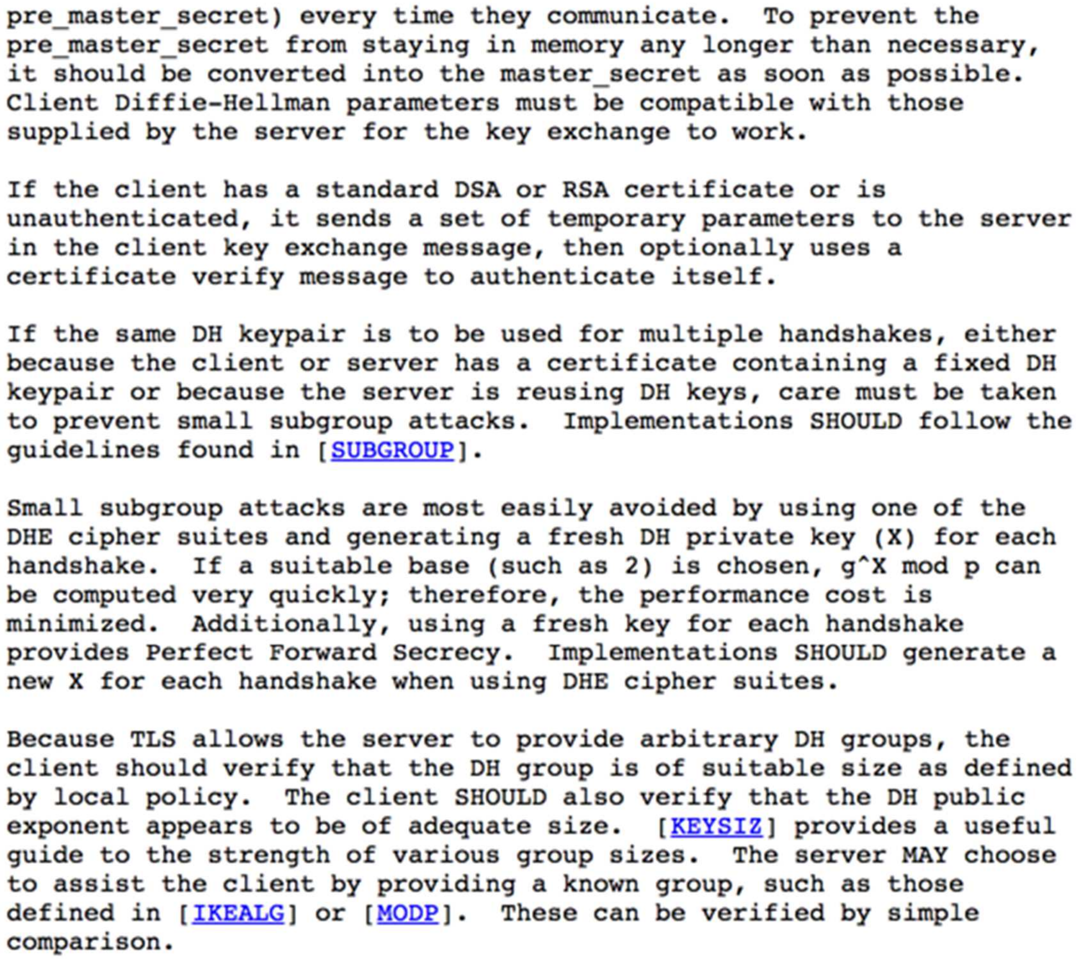
When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.



F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,

	 pre_master_secret) every time they communicate. To prevent the pre_master_secret from staying in memory any longer than necessary, it should be converted into the master_secret as soon as possible. Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work. If the client has a standard DSA or RSA certificate or is unauthenticated, it sends a set of temporary parameters to the server in the client key exchange message, then optionally uses a certificate verify message to authenticate itself. If the same DH keypair is to be used for multiple handshakes, either because the client or server has a certificate containing a fixed DH keypair or because the server is reusing DH keys, care must be taken to prevent small subgroup attacks. Implementations SHOULD follow the guidelines found in [SUBGROUP]. Small subgroup attacks are most easily avoided by using one of the DHE cipher suites and generating a fresh DH private key (X) for each handshake. If a suitable base (such as 2) is chosen, g^X mod p can be computed very quickly; therefore, the performance cost is minimized. Additionally, using a fresh key for each handshake provides Perfect Forward Secrecy. Implementations SHOULD generate a new X for each handshake when using DHE cipher suites. Because TLS allows the server to provide arbitrary DH groups, the client should verify that the DH group is of suitable size as defined by local policy. The client SHOULD also verify that the DH public exponent appears to be of adequate size. [KEYSIZ] provides a useful guide to the strength of various group sizes. The server MAY choose to assist the client by providing a known group, such as those defined in [IKEALG] or [MODP]. These can be verified by simple comparison. Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020 For each of the above key exchange algorithms, Google and/or ASICS Digital encrypts all communication, including upload of the mobile app, using a master secret, rendering the communication resistant to observation during transit from ASICS Digital to Google.

For **RSA and RSA _PSK**, ASICS Digital encrypts a random premaster secret with Google's RSA public key and sends the encrypted premaster secret to Google. Google decrypts the premaster secret with its matched RSA private key. ASICS Digital and Google both use the premaster secret to compute a master secret which is then used by both ASICS Digital and Google to encrypt all subsequent communications between ASICS Digital and Google.

For the other key exchange algorithms, namely Diffie-Hellman based algorithms such as **DHE_RSA, ECDHE_RSA, DH_RSA, DHE_DSS, DH_DSS, ECDH_RSA**, **ECDH_ECDSA and ECDHE_ECDSA,** ASICS Digital sends its Diffie-Hellman public key[17] to Google while Google sends its Diffie-Hellman public key to ASICS Digital. ASICS Digital then uses Google's Diffie-Hellman public key combined with ASICS Digital's own Diffie-Hellman private key to compute a premaster secret. Google in turn uses ASICS Digital's Diffie-Hellman public key combined with Google's own Diffie-Hellman private key to compute the same premaster secret. ASICS Digital and Google both use the premaster secret to compute a master secret which is then used by both ASICS Digital and Google to encrypt all subsequent communications between ASICS Digital and Google.

3. Resistant to Modification Because App Binary Is Code Signed

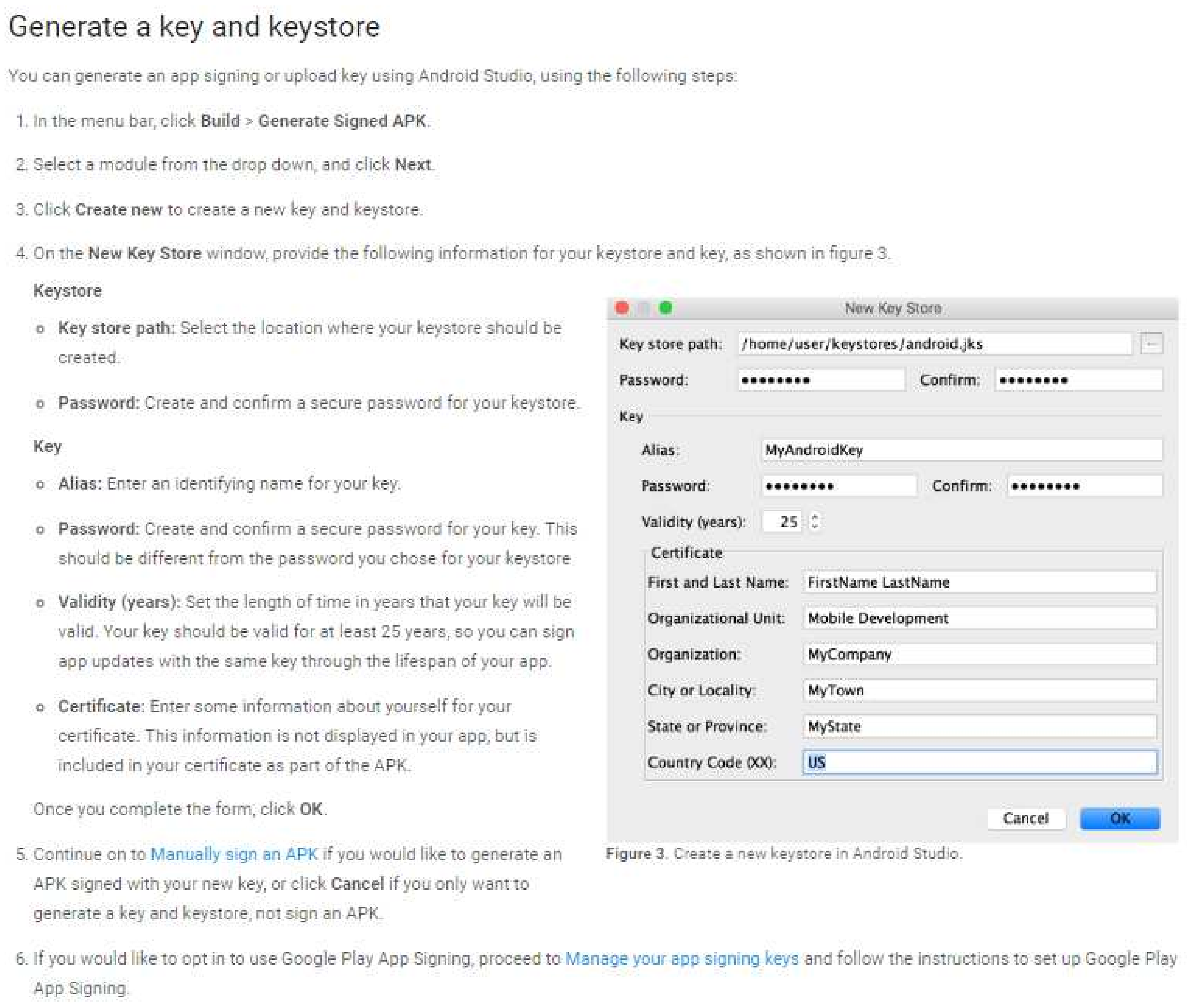
The mobile app code is made resistant to modification, at least in part, because the app binary is code signed. Google dictates that each developer must sign the mobile app submission with his/her asymmetric developer key that certifies that the app has not been modified by a third party impersonator[18].

> *Android requires that the user generates an asymmetric key all apps be digitally signed with a certificate before they can be installed. Android uses this certificate to identify the author of an app*
> Source: https://developer.android.com/studio/publish/app-signing.html, Last accessed on Mar 19, 2020
>
> *A public-key certificate, also known as a digital certificate or an identity certificate, contains the public key of a public/private key pair, as well as some other metadata identifying the owner of the key (for example, name and location). The owner of the certificate holds the corresponding private key.*
> *When you sign an APK, the signing tool attaches the public-key certificate to the APK. The public-key certificate serves as as a "fingerprint" that uniquely associates the APK to you and your corresponding private key. This helps Android ensure that any future updates to your APK are authentic and come from the original author.*
> *A keystore is a binary file that contains one or more private keys. When you sign an APK for release using Android Studio, you can choose to generate a new keystore and private key or use a keystore and private key you already have.*

[17] For Diffie-Hellman (DH) based algorithms such as DH_RSA, DHE_RSA, ECDH_RSA, ECDHE_RSA, DH_DSS, DHE_DSS, ECDH_ECDSA and ECDHE_ECDSA, Google calculates a hash of the message containing their Diffie-Hellman public key and encrypts the hash with their RSA/DSA/ECDSA private key (i.e. signing the message). Google then sends that RSA/DSA/ECDSA public key to ASICS Digital in a digital certificate so that ASICS Digital can authenticate the Google server by decrypting the hash using Google's public key and matching the decryption result to a hash of the received message as calculated by ASICS Digital itself. If the two values match, ASICS Digital knows that the message originated from Google and not from a malicious third party.

[18] *See, e.g.,* https://developer.android.com/tools/publishing/app-signing.html , Last accessed on Mar 19, 2020

Source: https://developer.android.com/studio/publish/app-signing.html , Last accessed on Mar 19, 2020



Generate a key and keystore

You can generate an app signing or upload key using Android Studio, using the following steps:

1. In the menu bar, click **Build** > **Generate Signed APK**.
2. Select a module from the drop down, and click **Next**.
3. Click **Create new** to create a new key and keystore.
4. On the **New Key Store** window, provide the following information for your keystore and key, as shown in figure 3.

   Keystore

   - **Key store path:** Select the location where your keystore should be created.
   - **Password:** Create and confirm a secure password for your keystore.

   Key

   - **Alias:** Enter an identifying name for your key.
   - **Password:** Create and confirm a secure password for your key. This should be different from the password you chose for your keystore
   - **Validity (years):** Set the length of time in years that your key will be valid. Your key should be valid for at least 25 years, so you can sign app updates with the same key through the lifespan of your app.
   - **Certificate:** Enter some information about yourself for your certificate. This information is not displayed in your app, but is included in your certificate as part of the APK.

   Once you complete the form, click **OK**.

5. Continue on to Manually sign an APK if you would like to generate an APK signed with your new key, or click **Cancel** if you only want to generate a key and keystore, not sign an APK.
6. If you would like to opt in to use Google Play App Signing, proceed to Manage your app signing keys and follow the instructions to set up Google Play App Signing.

New Key Store

Key store path: /home/user/keystores/android.jks

Password: •••••••• Confirm: ••••••••

Key

Alias: MyAndroidKey

Password: •••••••• Confirm: ••••••••

Validity (years): 25

Certificate

First and Last Name: FirstName LastName

Organizational Unit: Mobile Development

Organization: MyCompany

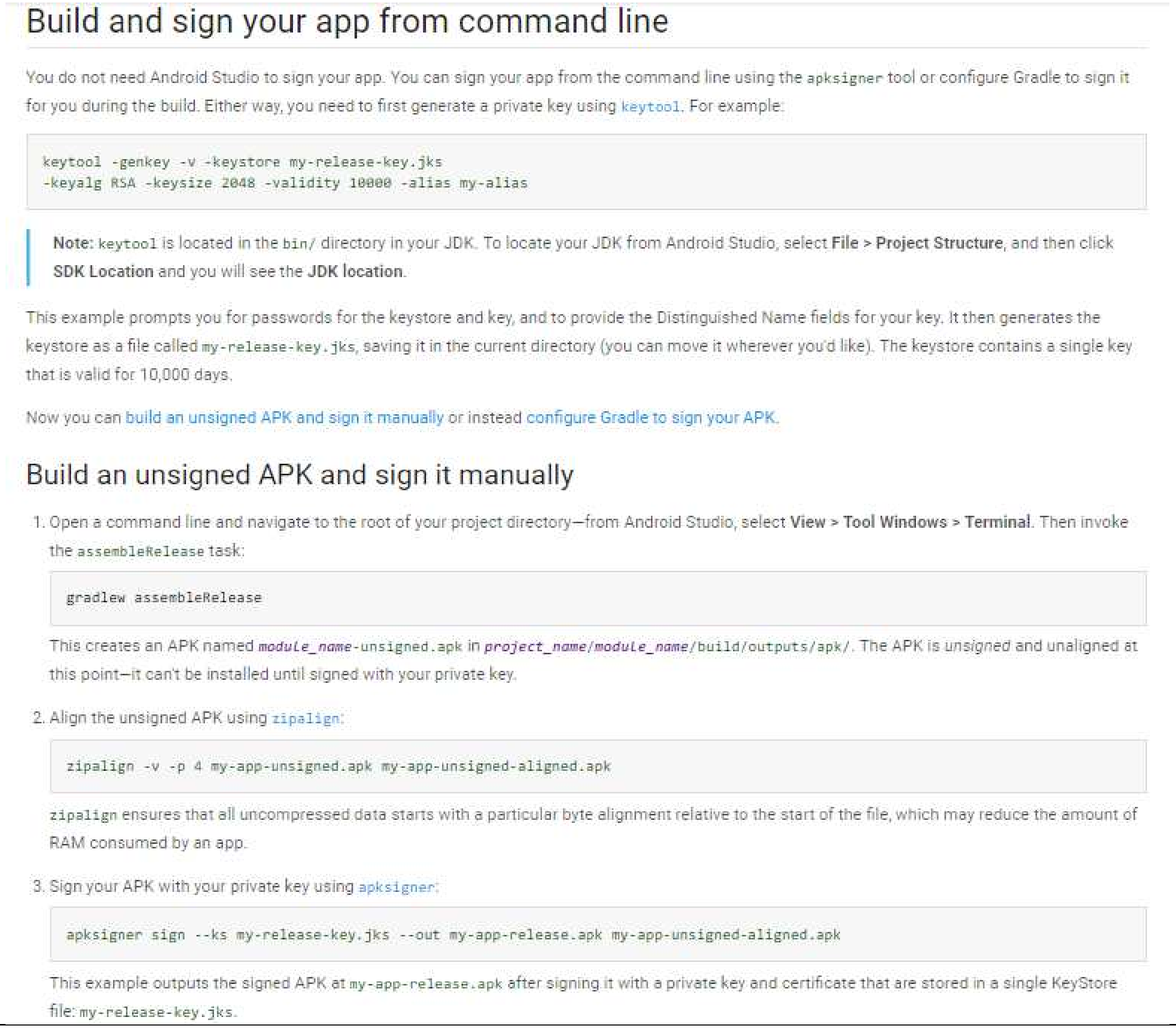
City or Locality: MyTown

State or Province: MyState

Country Code (XX): US

Figure 3. Create a new keystore in Android Studio.

Source: http://web.archive.org/web/20171107004101/https://developer.android.com/studio/publish/app-signing.html#sign-apk, Last accessed on Mar 19, 2020



Build and sign your app from command line

You do not need Android Studio to sign your app. You can sign your app from the command line using the `apksigner` tool or configure Gradle to sign it for you during the build. Either way, you need to first generate a private key using `keytool`. For example:

```
keytool -genkey -v -keystore my-release-key.jks
-keyalg RSA -keysize 2048 -validity 10000 -alias my-alias
```

> **Note:** `keytool` is located in the `bin/` directory in your JDK. To locate your JDK from Android Studio, select **File > Project Structure**, and then click **SDK Location** and you will see the **JDK location**.

This example prompts you for passwords for the keystore and key, and to provide the Distinguished Name fields for your key. It then generates the keystore as a file called `my-release-key.jks`, saving it in the current directory (you can move it wherever you'd like). The keystore contains a single key that is valid for 10,000 days.

Now you can build an unsigned APK and sign it manually or instead configure Gradle to sign your APK.

Build an unsigned APK and sign it manually

1. Open a command line and navigate to the root of your project directory—from Android Studio, select **View > Tool Windows > Terminal**. Then invoke the `assembleRelease` task:

```
gradlew assembleRelease
```

This creates an APK named `module_name-unsigned.apk` in `project_name/module_name/build/outputs/apk/`. The APK is *unsigned* and unaligned at this point—it can't be installed until signed with your private key.

2. Align the unsigned APK using `zipalign`:

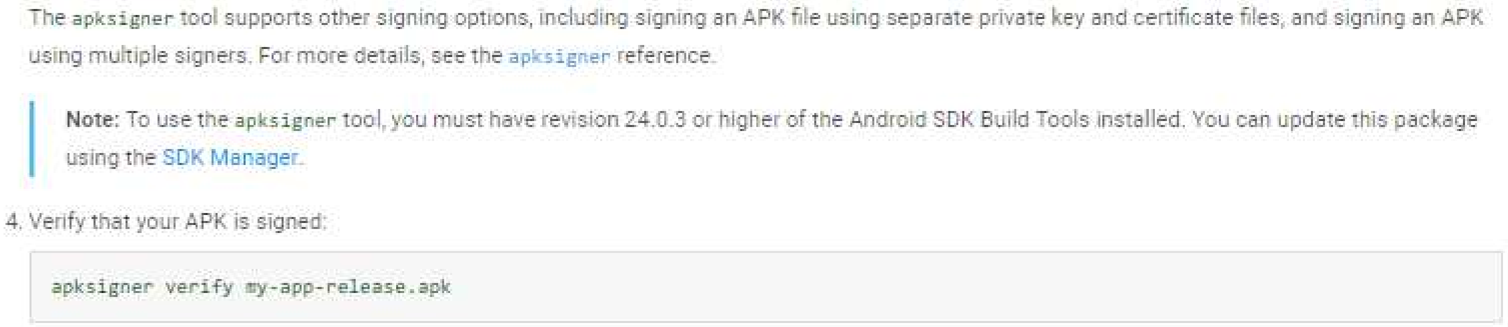
```
zipalign -v -p 4 my-app-unsigned.apk my-app-unsigned-aligned.apk
```

`zipalign` ensures that all uncompressed data starts with a particular byte alignment relative to the start of the file, which may reduce the amount of RAM consumed by an app.

3. Sign your APK with your private key using `apksigner`:

```
apksigner sign --ks my-release-key.jks --out my-app-release.apk my-app-unsigned-aligned.apk
```

This example outputs the signed APK at `my-app-release.apk` after signing it with a private key and certificate that are stored in a single KeyStore file: `my-release-key.jks`.



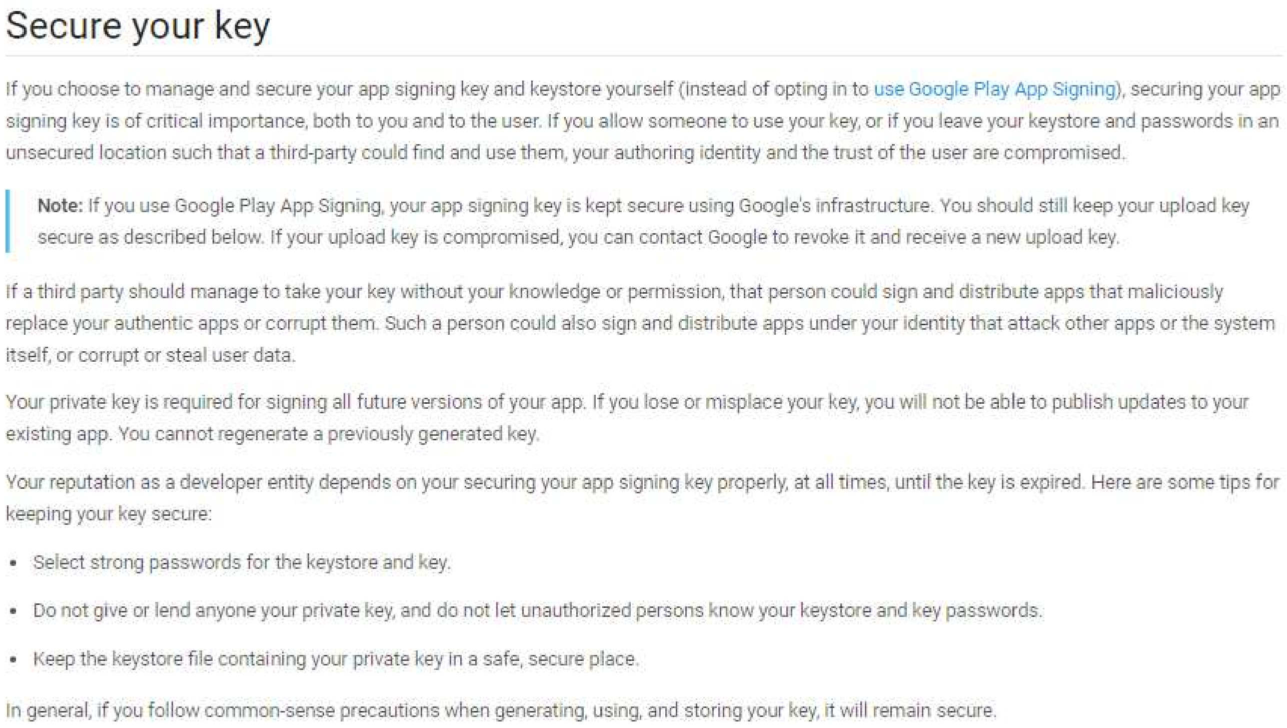
The apksigner tool supports other signing options, including signing an APK file using separate private key and certificate files, and signing an APK using multiple signers. For more details, see the apksigner reference.

> **Note:** To use the apksigner tool, you must have revision 24.0.3 or higher of the Android SDK Build Tools installed. You can update this package using the SDK Manager.

4. Verify that your APK is signed:

```
apksigner verify my-app-release.apk
```

Source: http://web.archive.org/web/20171107004101/https://developer.android.com/studio/publish/app-signing.html#sign-apk, Last accessed on Mar 19, 2020



Secure your key

If you choose to manage and secure your app signing key and keystore yourself (instead of opting in to use Google Play App Signing), securing your app signing key is of critical importance, both to you and to the user. If you allow someone to use your key, or if you leave your keystore and passwords in an unsecured location such that a third-party could find and use them, your authoring identity and the trust of the user are compromised.

> **Note:** If you use Google Play App Signing, your app signing key is kept secure using Google's infrastructure. You should still keep your upload key secure as described below. If your upload key is compromised, you can contact Google to revoke it and receive a new upload key.

If a third party should manage to take your key without your knowledge or permission, that person could sign and distribute apps that maliciously replace your authentic apps or corrupt them. Such a person could also sign and distribute apps under your identity that attack other apps or the system itself, or corrupt or steal user data.

Your private key is required for signing all future versions of your app. If you lose or misplace your key, you will not be able to publish updates to your existing app. You cannot regenerate a previously generated key.

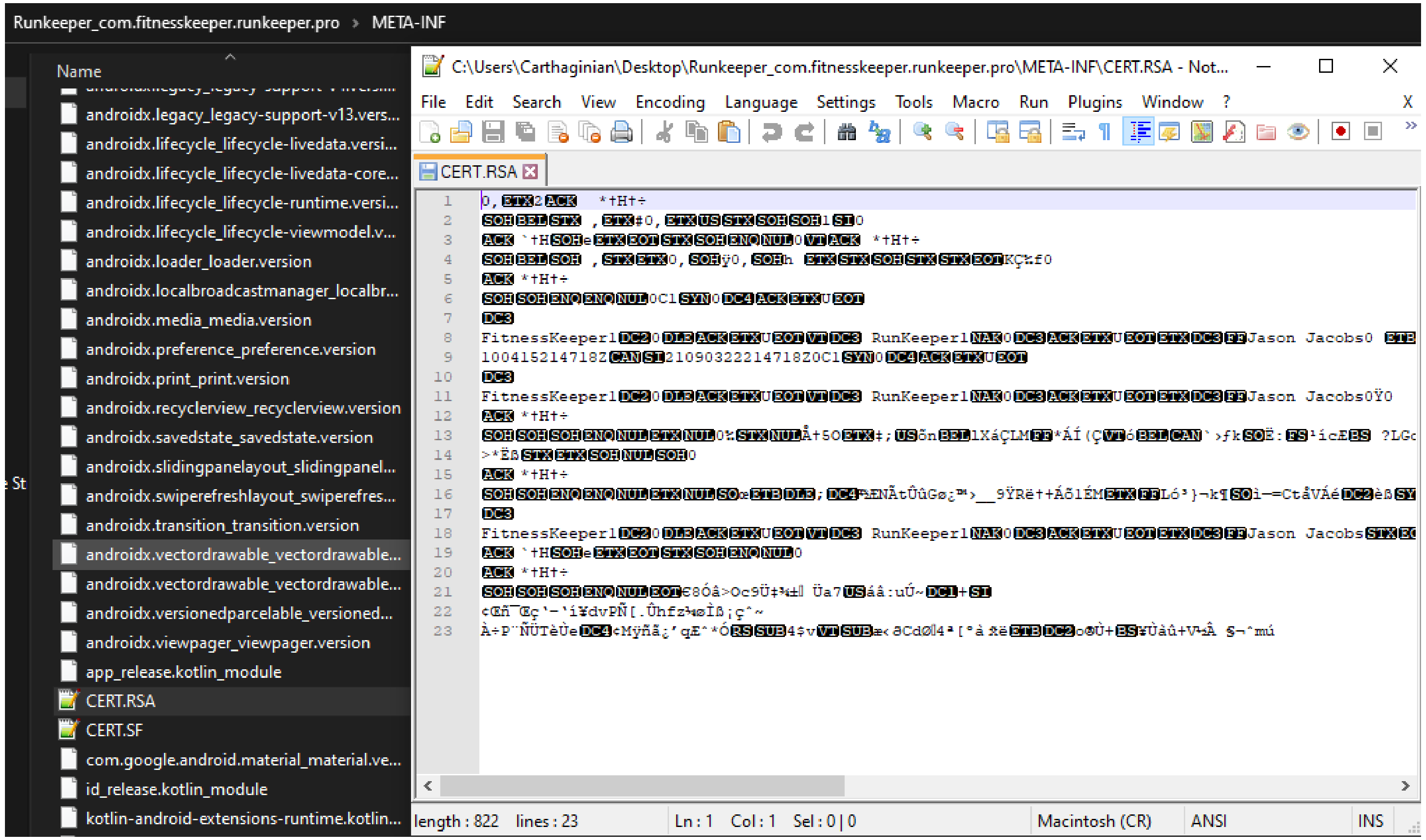
Your reputation as a developer entity depends on your securing your app signing key properly, at all times, until the key is expired. Here are some tips for keeping your key secure:

- Select strong passwords for the keystore and key.
- Do not give or lend anyone your private key, and do not let unauthorized persons know your keystore and key passwords.
- Keep the keystore file containing your private key in a safe, secure place.

In general, if you follow common-sense precautions when generating, using, and storing your key, it will remain secure.

Source: http://web.archive.org/web/20171107004101/https://developer.android.com/studio/publish/app-signing.html#sign-apk, Last accessed on Mar 19, 2020

ASICS Digital complies with Google's instructions on code signing as shown by ASICS Digital's mobile app contents. ASICS Digital's mobile apps contain files such as the files CERT.SF and CERT.RSA in ASICS Digital's Android apps which are generated during the code signing process as per instructions from Google.

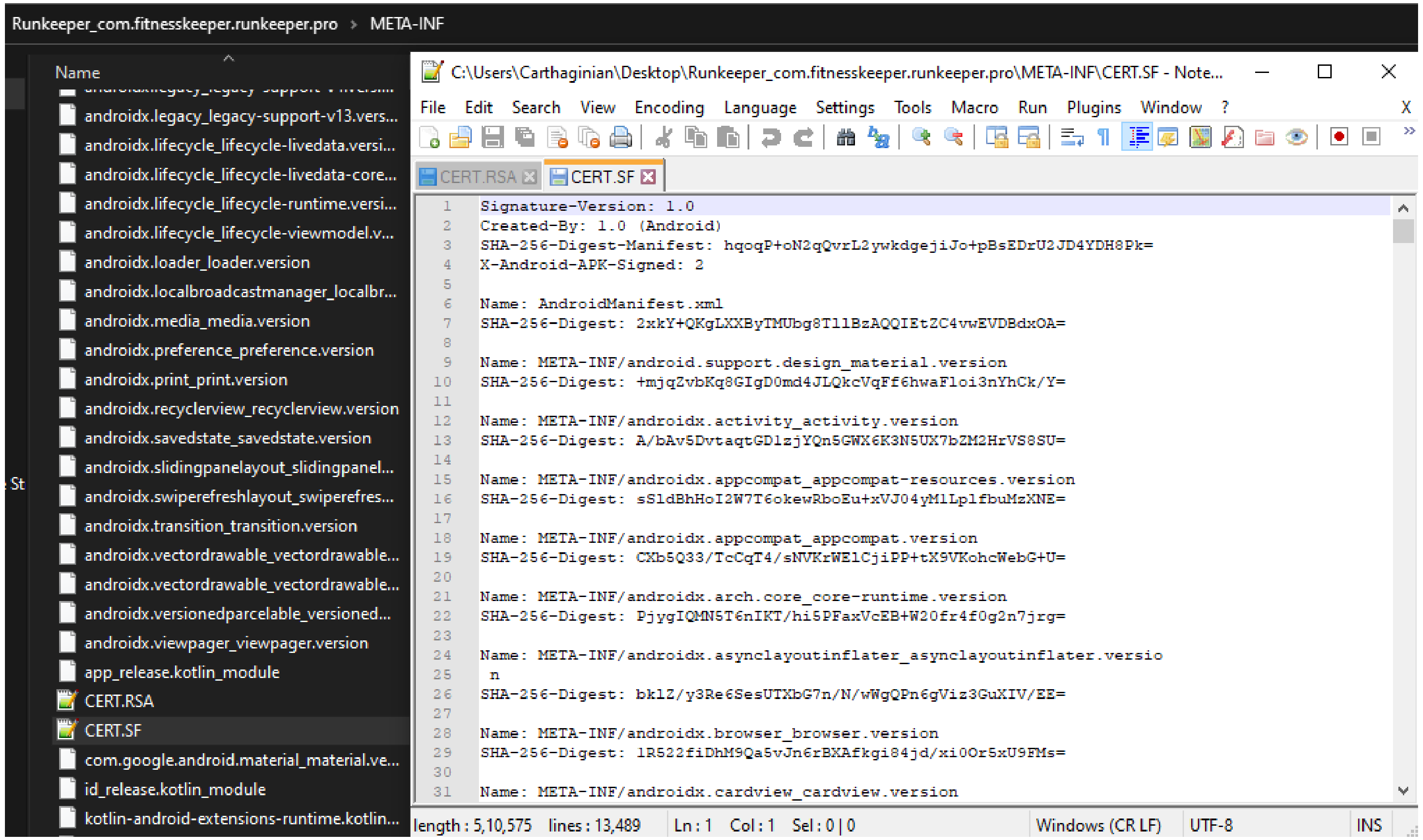

Runkeeper_com.fitnesskeeper.runkeeper.pro › META-INF
Name
C:\Users\Carthaginian\Desktop\Runkeeper_com.fitnesskeeper.runkeeper.pro\META-INF\CERT.RSA - Not...
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
CERT.RSA
St

androidx.legacy_legacy-support-v13.vers...
androidx.lifecycle_lifecycle-livedata.versi...
androidx.lifecycle_lifecycle-livedata-core...
androidx.lifecycle_lifecycle-runtime.versi...
androidx.lifecycle_lifecycle-viewmodel.v...
androidx.loader_loader.version
androidx.localbroadcastmanager_localbr...
androidx.media_media.version
androidx.preference_preference.version
androidx.print_print.version
androidx.recyclerview_recyclerview.version
androidx.savedstate_savedstate.version
androidx.slidingpanelayout_slidingpanel...
androidx.swiperefreshlayout_swiperefres...
androidx.transition_transition.version
androidx.vectordrawable_vectordrawable...
androidx.vectordrawable_vectordrawable...
androidx.versionedparcelable_versioned...
androidx.viewpager_viewpager.version
app_release.kotlin_module
CERT.RSA
CERT.SF
com.google.android.material_material.ve...
id_release.kotlin_module
kotlin-android-extensions-runtime.kotlin...

```
0,ETX2ACK   *†H†÷
SOHBELSTX ,ETX#0,ETXUSSTXSOHSOH1SI0
ACK `†HSOHeETXEOTSTXSOHENQNUL0VTACK *†H†÷
SOHBELSOH ,STXETX0,SOHÿ0,SOHh ETXSTXSOHSTXSTXEOTKÇ%f0
ACK *†H†÷
SOHSOHENQENQNUL0C1SYN0DC4ACKETXUEOT
DC3
FitnessKeeper1DC20DLEACKETXUEOTVTDC3 RunKeeper1NAK0DC3ACKETXUEOTETXDC3FFJason Jacobs0 ETB
100415214718ZCANSI210903222147182Z0C1SYN0DC4ACKETXUEOT
DC3
FitnessKeeper1DC20DLEACKETXUEOTVTDC3 RunKeeper1NAK0DC3ACKETXUEOTETXDC3FFJason Jacobs0Ÿ0
ACK *†H†÷
SOHSOHSOHENQNULETXNUL0%STXNULÅ+5OETX‡;USõnBEL1XáÇLMFF*ÁÍ(ÇVTóBELCAN`›fkSOË:FS¹icÆBS ?LGc
>*ËßSTXETXSOHNULSOH0
ACK *†H†÷
SOHSOHENQENQNULETXNULSOæETBDLE;DC4ÆENÃtÛûGø¿™›__9ŸRë†+Áõ1ÉMETXFFLó³}¬k¶SOì—=CtåVÁéDC2èßSY
DC3
FitnessKeeper1DC20DLEACKETXUEOTVTDC3 RunKeeper1NAK0DC3ACKETXUEOTETXDC3FFJason JacobsSTXEO
ACK `†HSOHeETXEOTSTXSOHENQNUL0
ACK *†H†÷
SOHSOHSOHENQNULEOT€8Óâ>Oc9Ü‡¾±‖ Üa7USáâ:uÛ~DC1+SI
¢Œñ¯Œç'–'i¥dvPÑ[.Ûhfz¼øÌß;ç^~
À÷P¨ÑÜTèÙeDC4¢Mÿñã¿'qE^*ÓRSSUB4$vVTSUBæ‹ ðCdØ‖4ª[°à ‰ëETBDC2o®Ù+BS¥Ùàû+V¼Â $¬^mú
```

length : 822 lines : 23 Ln : 1 Col : 1 Sel : 0 | 0 Macintosh (CR) ANSI INS

Source: Contents of Runkeeper - GPS Track Run Walk (https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro, Last accessed on Mar 19, 2020) as an example of ASICS Digital app


Runkeeper_com.fitnesskeeper.runkeeper.pro › META-INF
Name
C:\Users\Carthaginian\Desktop\Runkeeper_com.fitnesskeeper.runkeeper.pro\META-INF\CERT.SF - Note...
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
CERT.RSA
CERT.SF
St

androidx.legacy_legacy-support-v13.vers...
androidx.lifecycle_lifecycle-livedata.versi...
androidx.lifecycle_lifecycle-livedata-core...
androidx.lifecycle_lifecycle-runtime.versi...
androidx.lifecycle_lifecycle-viewmodel.v...
androidx.loader_loader.version
androidx.localbroadcastmanager_localbr...
androidx.media_media.version
androidx.preference_preference.version
androidx.print_print.version
androidx.recyclerview_recyclerview.version
androidx.savedstate_savedstate.version
androidx.slidingpanelayout_slidingpanel...
androidx.swiperefreshlayout_swiperefres...
androidx.transition_transition.version
androidx.vectordrawable_vectordrawable...
androidx.vectordrawable_vectordrawable...
androidx.versionedparcelable_versioned...
androidx.viewpager_viewpager.version
app_release.kotlin_module
CERT.RSA
CERT.SF
com.google.android.material_material.ve...
id_release.kotlin_module
kotlin-android-extensions-runtime.kotlin...

```
Signature-Version: 1.0
Created-By: 1.0 (Android)
SHA-256-Digest-Manifest: hqoqP+oN2qQvrL2ywkdgejiJo+pBsEDrU2JD4YDH8Pk=
X-Android-APK-Signed: 2

Name: AndroidManifest.xml
SHA-256-Digest: 2xkY+QKgLXXByTMUbg8T1lBzAQQIEtZC4vwEVDBdxOA=

Name: META-INF/android.support.design_material.version
SHA-256-Digest: +mjqZvbKq8GIgD0md4JLQkcVqFf6hwaFloi3nYhCk/Y=

Name: META-INF/androidx.activity_activity.version
SHA-256-Digest: A/bAv5DvtaqtGDlzjYQn5GWX6K3N5UX7bZM2HrVS8SU=

Name: META-INF/androidx.appcompat_appcompat-resources.version
SHA-256-Digest: sSldBhHoI2W7T6okewRboEu+xVJ04yMlLplfbuMzXNE=

Name: META-INF/androidx.appcompat_appcompat.version
SHA-256-Digest: CXb5Q33/TcCqT4/sNVKrWElCjiPP+tX9VKohcWebG+U=

Name: META-INF/androidx.arch.core_core-runtime.version
SHA-256-Digest: PjygIQMN5T6nIKT/hi5PFaxVcEB+W20fr4f0g2n7jrg=

Name: META-INF/androidx.asynclayoutinflater_asynclayoutinflater.versio
 n
SHA-256-Digest: bklZ/y3Re6SesUTXbG7n/N/wWgQPn6gViz3GuXIV/EE=

Name: META-INF/androidx.browser_browser.version
SHA-256-Digest: 1R522fiDhM9Qa5vJn6rBXAfkgi84jd/xi0Or5xU9FMs=

Name: META-INF/androidx.cardview_cardview.version
```

length : 5,10,575 lines : 13,489 Ln : 1 Col : 1 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

Source: Contents of Runkeeper - GPS Track Run Walk (https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro, Last accessed on Mar 19, 2020) as an example of ASICS Digital app

	Asymmetrical key cryptography and hashing algorithms are used to create the unique digital signature for Android mobile apps. The digital signature is used to sign the resources in an application package, including the compiled code. The private key of an asymmetric key pair that is generated for the digital code signing is used to code sign the app. This private key is included in the mobile app although the private key is not the claimed private key of the claimed generated asymmetric key pair because it does not match the claimed public key used to encrypt predetermined data, based on the court's construction. Hashes are created for every resource in the application package with the help of a hash algorithm. The signature manifest also has its own hash to prevent unauthorized changes. The hashes are encrypted with a private key. After the encryption is complete, the digital signature for the app is created. By signing the app binary with a digital signature, ASICS Digital's mobile apps are tamper resistant enabling Google and the Android mobile devices to verify that the application is being distributed by trusted source (*i.e.* ASICS Digital) and that the application has not been modified by a third party, which can be verified by the corresponding public key generated as part of the pair. Thus the app binary is made resistant to modification by digital signing. Accordingly ASICS Digital's Android mobile apps establish SSL/TLS communications with ASICS Digital's servers, which involve a SSL/TLS handshake procedure involving asymmetric key encryption. SSL/TLS ensures secure communication and renders the mobile app data further resistant to observation.
2. The method of claim 1, wherein the program is on a remote system and further comprising sending the executable tamper resistant key module to the remote system.	ASICS Digital's mobile software application products and services including by way of example, but not limited to the following apps ("mobile applications", "mobile apps" or "Accused Products") that are specifically developed, used, sold, offered for sale, marketed, licensed and distributed by ASICS Digital to be downloaded onto Android mobile or tablet devices. • Runkeeper - GPS Track Run Walk (https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro, Last accessed on Mar 19, 2020) • ASICS Studio: Run, Lift & Get Fit (https://play.google.com/store/apps/details?id=com.fitnesskeeper.asicsstudio, Last accessed on Mar 19, 2020) ASICS Digital directly infringes and/or continues to knowingly induce Google to infringe this claim by intentionally developing, making, marketing, advertising, providing, sending, distributing and licensing its mobile applications software, documentation, materials, training or support and aiding, abetting, encouraging, promoting or inviting use thereof. Upon information and belief, the method step of sending the executable tamper resistant key module to the remote system is performed by Google and/or its agents – whose acts are attributable to ASICS Digital (i) because ASICS Digital works together with Google in a joint enterprise in the

building and distribution of its mobile apps, or (ii) because Google distributes and markets ASICS Digital's mobile apps under the direction and control of ASICS Digital, or acts as agent, or on behalf of ASICS Digital, in the building, marketing and distribution of ASICS Digital's mobile apps.

Alternatively, to the extent any portion of this method step is performed by ASICS Digital, such acts are attributable to Google, who conditions participation in and the receipt of a benefit, namely, the distribution of ASICS Digital's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Google in the building and upload of ASICS Digital's mobile apps, and ASICS Digital induces infringement by Google in the building, marketing and distribution of ASICS Digital's mobile apps.

ASICS Digital's mobile apps are sent or downloaded from Google servers and are executed on Android remote devices such as mobile phones and tablets. When a user accesses Google Play Store – and requests to download ASICS Digital app, Google sends the executable tamper resistant key module from the Google servers to the remote device(s).

Further, the step of "sending" ASICS Digital mobile apps to the remote system occurs via TLS/SSL communications. Thus, the sending of ASICS Digital mobile apps, to the extent required by the claims, also includes a private key and predetermined data encrypted by a public key, as explained in detail above.

In particular, ASICS Digital mobile apps sent to users' remote devices are tamper resistant, resistant to observation and modification as follows:

1. **Resistant to Observation Because App is Downloaded in Compiled Form**

ASICS Digital mobile apps are resistant to observation, at least in part, since ASICS Digital compiles its mobile app source code before submitting the app to Google – and uploads the binary output of the compilation process rather than the source code itself – and hence a user can only download the compiled source code from Google rather than the source code itself[19].

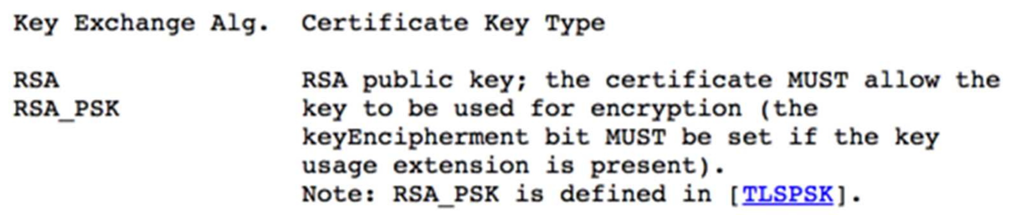
See, Android Studio Users Guide, "Prepare for Release" stating "To release your application to users you need to create a release-ready package that users can install and run on their Android-powered devices. The release-ready package contains the same components as the debug APK file — compiled source code, resources, manifest file, and so on — and it is built using the same build tools. However, unlike the debug APK file, the release-ready APK file is signed with your own certificate and it is optimized with the zipalign tool." https://developer.android.com/studio/publish/preparing.html, Last accessed on Mar 19, 2020

2. **Resistant to Observation Because Download from Google Is Over SSL/TLS**

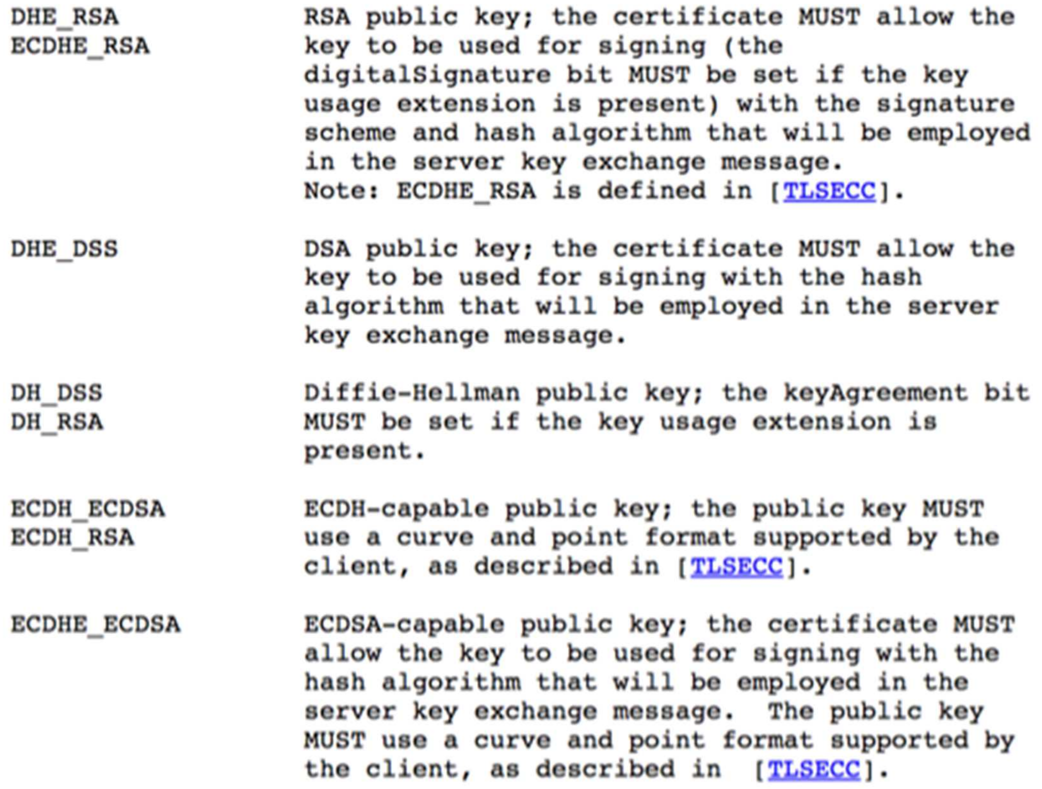
[19] *See, e.g.,* https://developer.android.com/studio/build/index.html, Last accessed on Mar 19, 2020

ASICS Digital's mobile apps are made further resistant to observation, at least in part, because the mobile app is securely sent or downloaded by SSL/TLS from Google servers. ASICS Digital app users establish SSL/TLS communications with Play Store (for example using the URL https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro for Runkeeper - GPS Track Run Walk, Last accessed on Mar 19, 2020) when downloading ASICS Digital's apps, as evidenced by the "https" in the URL. Sending the mobile app code by SSL/TLS is necessary to keep the code from being observed in transit from Google to the user's remote system.

The secure download process starts with a TLS handshake procedure which uses asymmetric key encryption and necessitates generation of at least one asymmetric key pair. Specifically, user's remote device negotiates with Google the cipher suite and the key exchange algorithm that will be used for the handshake:



```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].
```



```
DHE_RSA              RSA public key; the certificate MUST allow the
ECDHE_RSA            key to be used for signing (the
                     digitalSignature bit MUST be set if the key
                     usage extension is present) with the signature
                     scheme and hash algorithm that will be employed
                     in the server key exchange message.
                     Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS              DSA public key; the certificate MUST allow the
                     key to be used for signing with the hash
                     algorithm that will be employed in the server
                     key exchange message.

DH_DSS               Diffie-Hellman public key; the keyAgreement bit
DH_RSA               MUST be set if the key usage extension is
                     present.

ECDH_ECDSA           ECDH-capable public key; the public key MUST
ECDH_RSA             use a curve and point format supported by the
                     client, as described in [TLSECC].

ECDHE_ECDSA          ECDSA-capable public key; the certificate MUST
                     allow the key to be used for signing with the
                     hash algorithm that will be employed in the
                     server key exchange message.  The public key
                     MUST use a curve and point format supported by
                     the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

Each of these algorithms necessitates generating one or more asymmetric key pairs – that are in turn used to compute a shared master secret for encrypting the mobile app download.

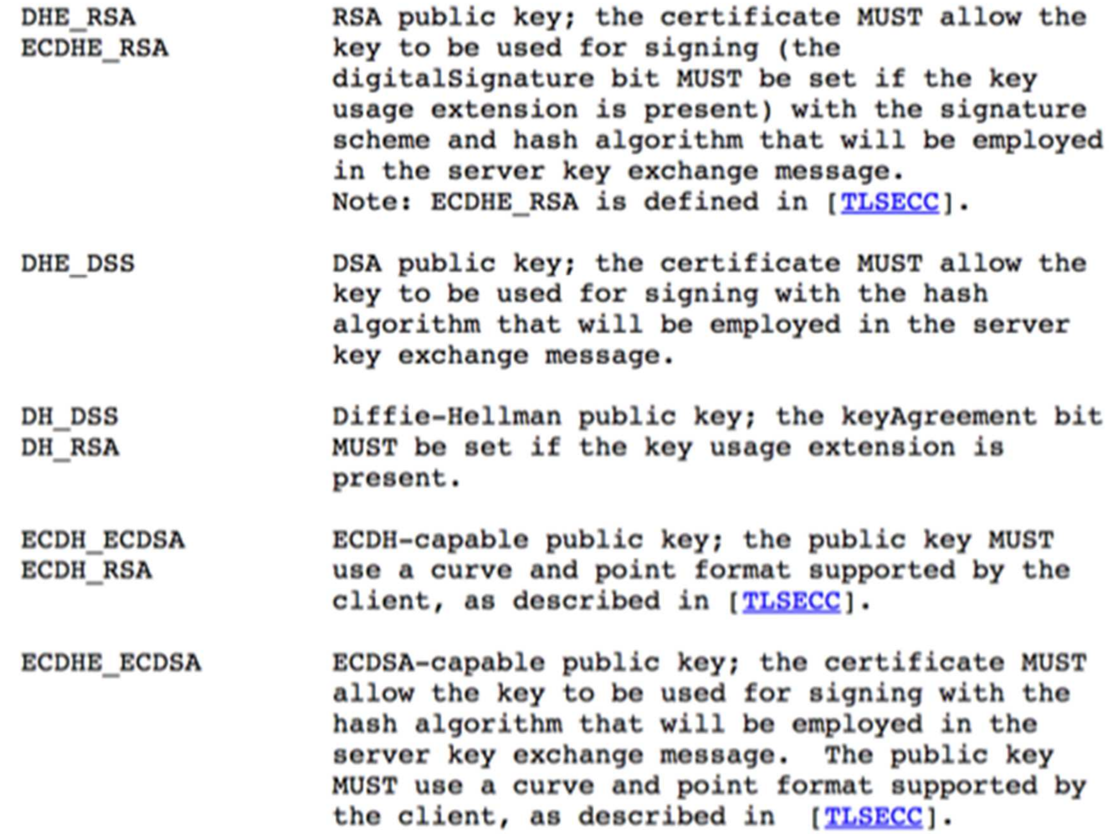
For **RSA and RSA _PSK**, Google server generates an RSA public-private key pair.

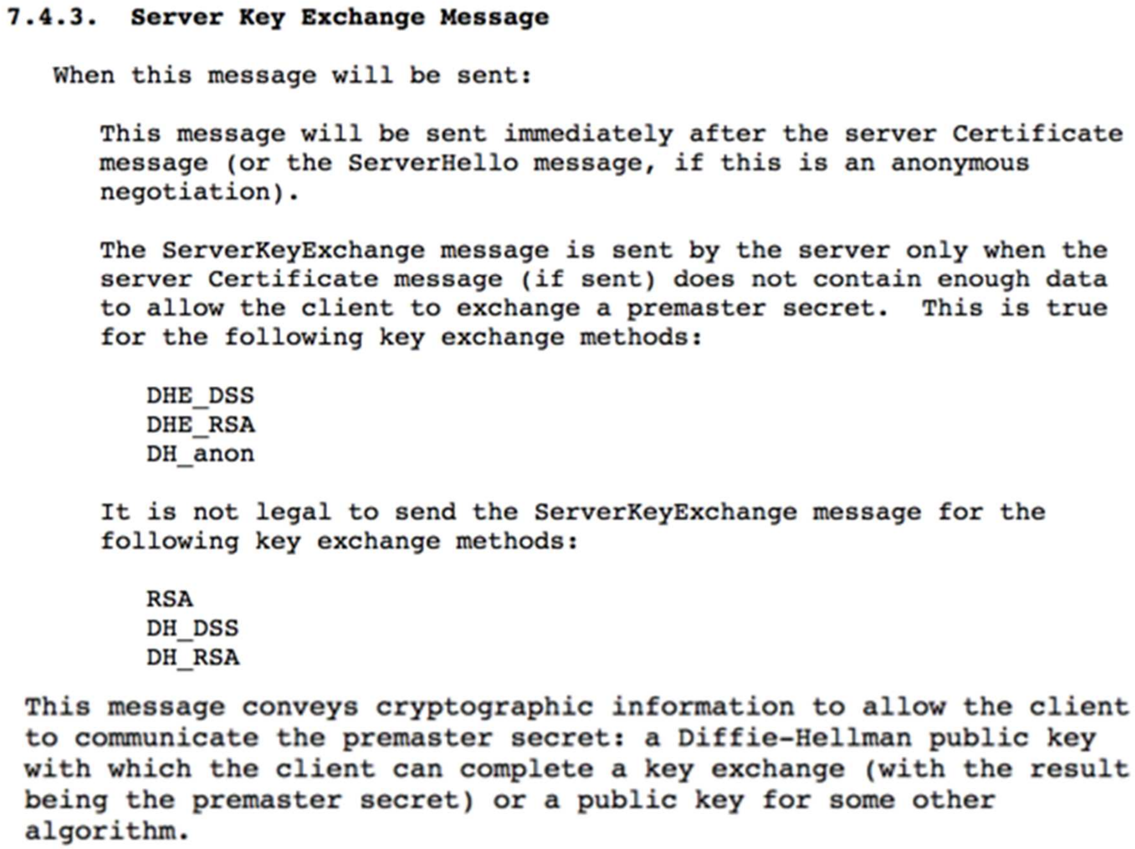
For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, Google server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[20]. The user's remote device also generates a second Diffie-Hellman public-private key pair.

[20] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf,

	For **DHE_DSS and DH_DSS**, Google server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. For **ECDH_ECDSA and ECDHE_ECDSA**, Google server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair.

http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN 1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020

	 DHE_RSA ECDHE_RSA RSA public key; the certificate MUST allow the key to be used for signing (the digitalSignature bit MUST be set if the key usage extension is present) with the signature scheme and hash algorithm that will be employed in the server key exchange message. Note: ECDHE_RSA is defined in [TLSECC]. DHE_DSS DSA public key; the certificate MUST allow the key to be used for signing with the hash algorithm that will be employed in the server key exchange message. DH_DSS DH_RSA Diffie-Hellman public key; the keyAgreement bit MUST be set if the key usage extension is present. ECDH_ECDSA ECDH_RSA ECDH-capable public key; the public key MUST use a curve and point format supported by the client, as described in [TLSECC]. ECDHE_ECDSA ECDSA-capable public key; the certificate MUST allow the key to be used for signing with the hash algorithm that will be employed in the server key exchange message. The public key MUST use a curve and point format supported by the client, as described in [TLSECC]. *Source: https://tools.ietf.org/html/rfc5246 at 48-49*, Last accessed on Mar 19, 2020



7.4.3. Server Key Exchange Message

When this message will be sent:

This message will be sent immediately after the server Certificate message (or the ServerHello message, if this is an anonymous negotiation).

The ServerKeyExchange message is sent by the server only when the server Certificate message (if sent) does not contain enough data to allow the client to exchange a premaster secret. This is true for the following key exchange methods:

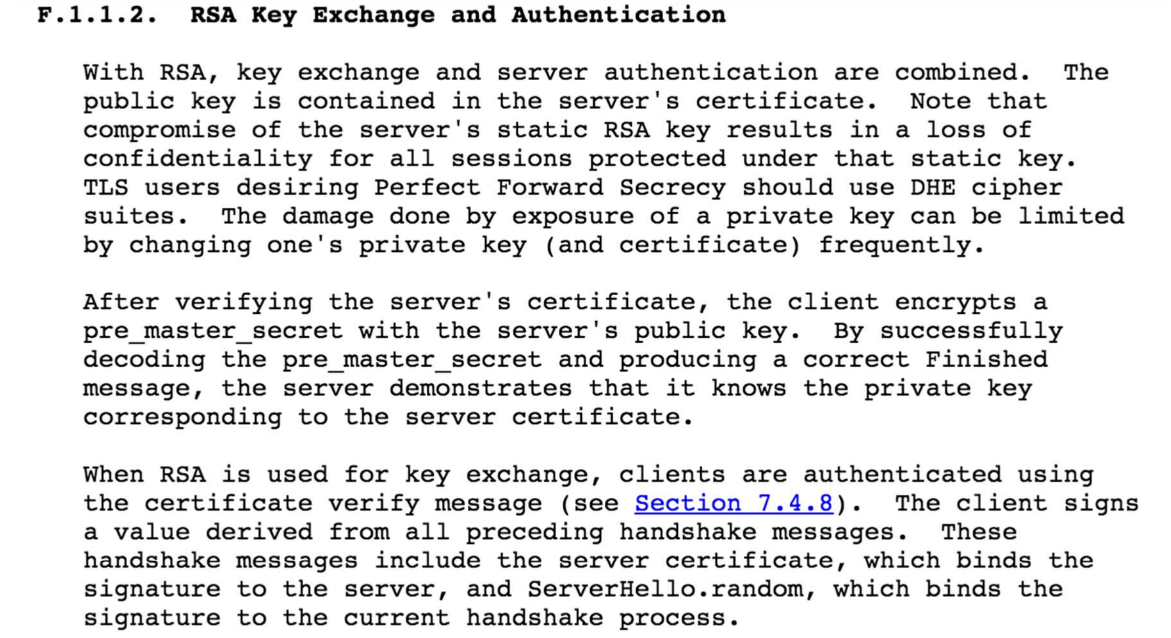
DHE_DSS
DHE_RSA
DH_anon

It is not legal to send the ServerKeyExchange message for the following key exchange methods:

RSA
DH_DSS
DH_RSA

This message conveys cryptographic information to allow the client to communicate the premaster secret: a Diffie-Hellman public key with which the client can complete a key exchange (with the result being the premaster secret) or a public key for some other algorithm.

Source: https://tools.ietf.org/html/rfc5246 at 50-51, Last accessed on Mar 19, 2020

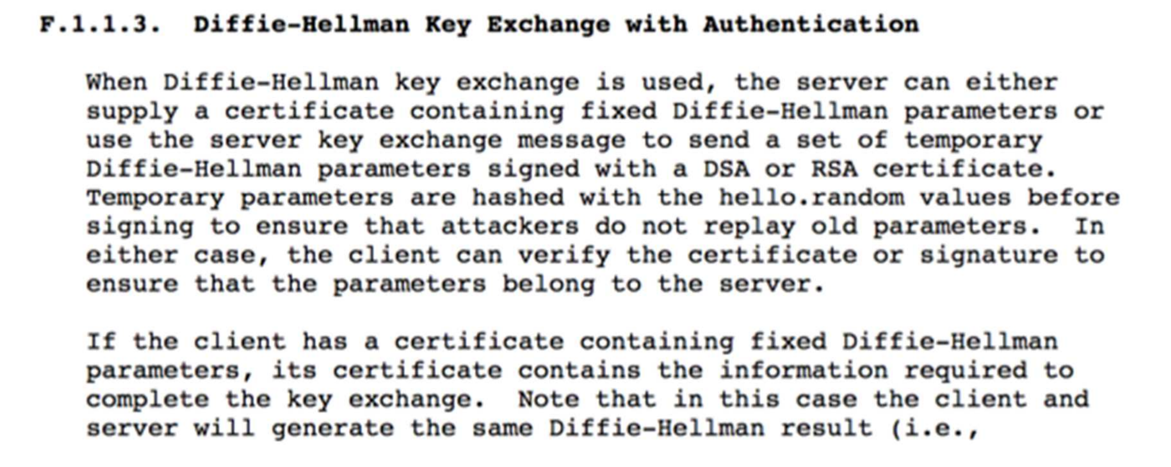


F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

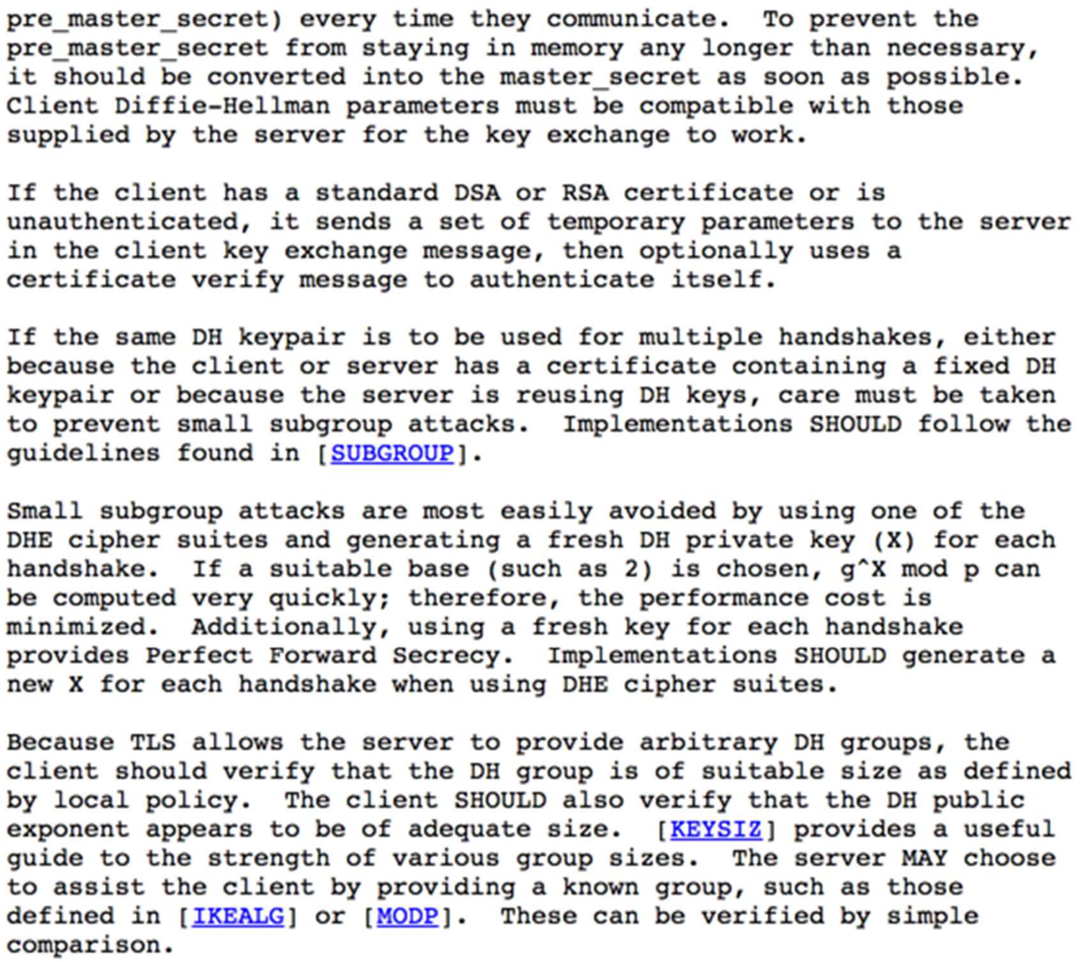
When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.



F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,

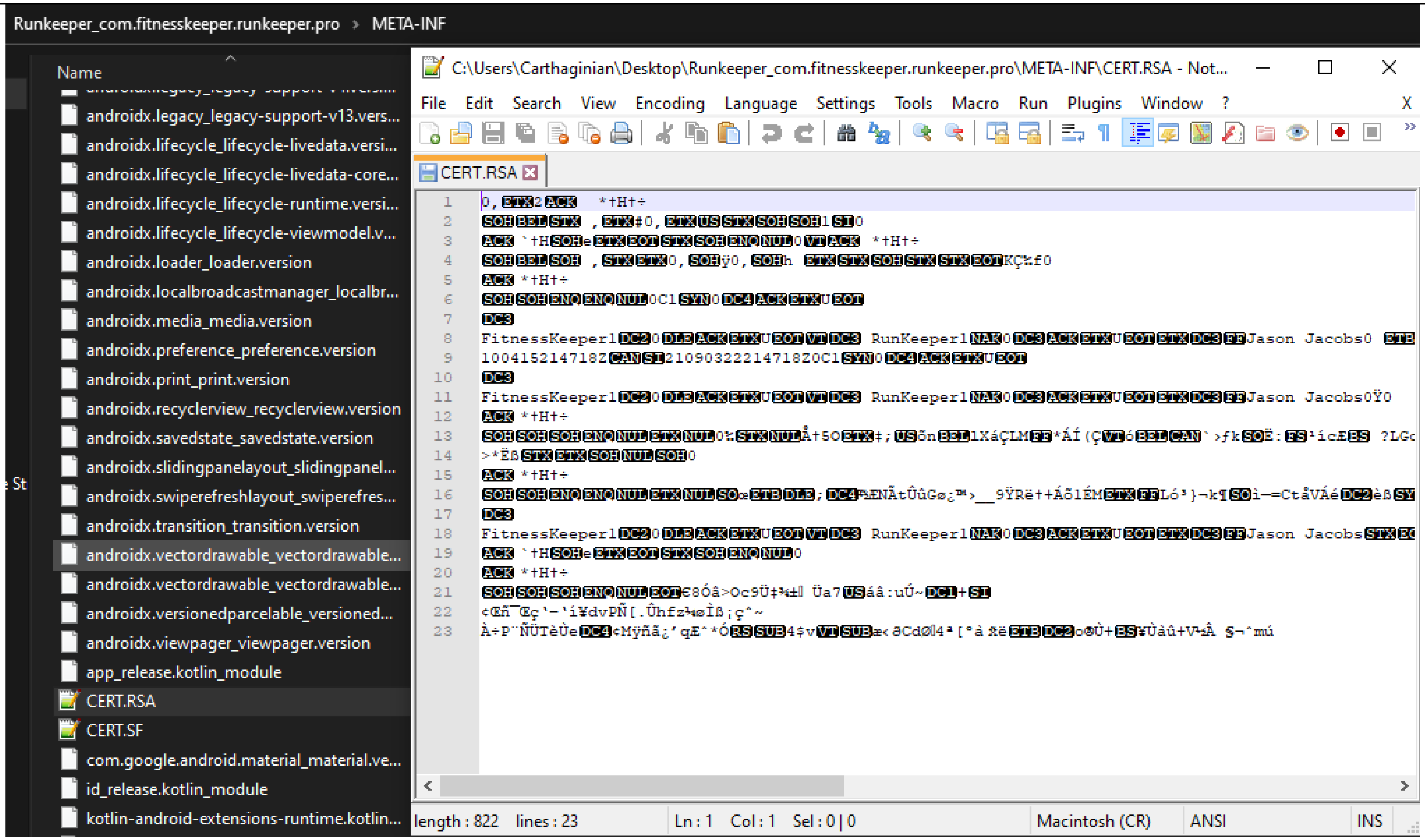
	 pre_master_secret) every time they communicate. To prevent the pre_master_secret from staying in memory any longer than necessary, it should be converted into the master_secret as soon as possible. Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work. If the client has a standard DSA or RSA certificate or is unauthenticated, it sends a set of temporary parameters to the server in the client key exchange message, then optionally uses a certificate verify message to authenticate itself. If the same DH keypair is to be used for multiple handshakes, either because the client or server has a certificate containing a fixed DH keypair or because the server is reusing DH keys, care must be taken to prevent small subgroup attacks. Implementations SHOULD follow the guidelines found in [SUBGROUP]. Small subgroup attacks are most easily avoided by using one of the DHE cipher suites and generating a fresh DH private key (X) for each handshake. If a suitable base (such as 2) is chosen, g^X mod p can be computed very quickly; therefore, the performance cost is minimized. Additionally, using a fresh key for each handshake provides Perfect Forward Secrecy. Implementations SHOULD generate a new X for each handshake when using DHE cipher suites. Because TLS allows the server to provide arbitrary DH groups, the client should verify that the DH group is of suitable size as defined by local policy. The client SHOULD also verify that the DH public exponent appears to be of adequate size. [KEYSIZ] provides a useful guide to the strength of various group sizes. The server MAY choose to assist the client by providing a known group, such as those defined in [IKEALG] or [MODP]. These can be verified by simple comparison. Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020 The generated asymmetric key pairs are then used to compute a shared master secret which is then used to encrypt the mobile app download so that it is resistant to observation during transit.

	For **RSA and RSA _PSK**, the RSA public-private key pair is used to encrypt a random premaster secret which is in turn used by Google server and the user's remote device to compute a master secret. Google uses the master secret to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, Google server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[21]. The user's remote device also generates a second Diffie-Hellman public-private key pair. Google server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Google's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret that Google uses to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. For **DHE_DSS and DH_DSS**, Google server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. Google server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Google's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret that Google uses to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. For **ECDH_ECDSA and ECDHE_ECDSA**, Google server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. Google server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Google's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret that Google uses to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. **3. Resistant to Modification Because Mobile App is Code Signed** The downloaded mobile app code is resistant to modification, at least in part, because the downloaded app binary is code signed. Code-signing allows users' remote systems to verify that the downloaded app binary is authentic and has not been maliciously modified by a third party. Google

[21] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN
1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020

	dictates that each developer must sign the mobile app submission with his/her asymmetric developer key that certifies that the app has not been modified by a third party impersonator[22]. *Android requires that the user generates an asymmetric key all apps be digitally signed with a certificate before they can be installed. Android uses this certificate to identify the author of an app* Source: https://developer.android.com/studio/publish/app-signing.html, Last accessed on Mar 19, 2020 *A public-key certificate, also known as a digital certificate or an identity certificate, contains the public key of a public/private key pair, as well as some other metadata identifying the owner of the key (for example, name and location). The owner of the certificate holds the corresponding private key.* *When you sign an APK, the signing tool attaches the public-key certificate to the APK. The public-key certificate serves as as a "fingerprint" that uniquely associates the APK to you and your corresponding private key. This helps Android ensure that any future updates to your APK are authentic and come from the original author.* *A keystore is a binary file that contains one or more private keys. When you sign an APK for release using Android Studio, you can choose to generate a new keystore and private key or use a keystore and private key you already have.* Source: https://developer.android.com/studio/publish/app-signing.html , Last accessed on Mar 19, 2020 ASICS Digital complies with Google's instructions on code signing as shown by ASICS Digital's mobile app contents. ASICS Digital's mobile apps contain files such as the files CERT.SF and CERT.RSA in ASICS Digital's Android mobile apps which are generated during the code signing process as per instructions from Google.

[22] *See, e.g.,* https://developer.android.com/tools/publishing/app-signing.html, Last accessed on Mar 19, 2020



Runkeeper_com.fitnesskeeper.runkeeper.pro › META-INF

Name
androidx.legacy_legacy-support-v13.vers...
androidx.lifecycle_lifecycle-livedata.versi...
androidx.lifecycle_lifecycle-livedata-core...
androidx.lifecycle_lifecycle-runtime.versi...
androidx.lifecycle_lifecycle-viewmodel.v...
androidx.loader_loader.version
androidx.localbroadcastmanager_localbr...
androidx.media_media.version
androidx.preference_preference.version
androidx.print_print.version
androidx.recyclerview_recyclerview.version
androidx.savedstate_savedstate.version
androidx.slidingpanelayout_slidingpanel...
androidx.swiperefreshlayout_swiperefres...
androidx.transition_transition.version
androidx.vectordrawable_vectordrawable...
androidx.vectordrawable_vectordrawable...
androidx.versionedparcelable_versioned...
androidx.viewpager_viewpager.version
app_release.kotlin_module
CERT.RSA
CERT.SF
com.google.android.material_material.ve...
id_release.kotlin_module
kotlin-android-extensions-runtime.kotlin...

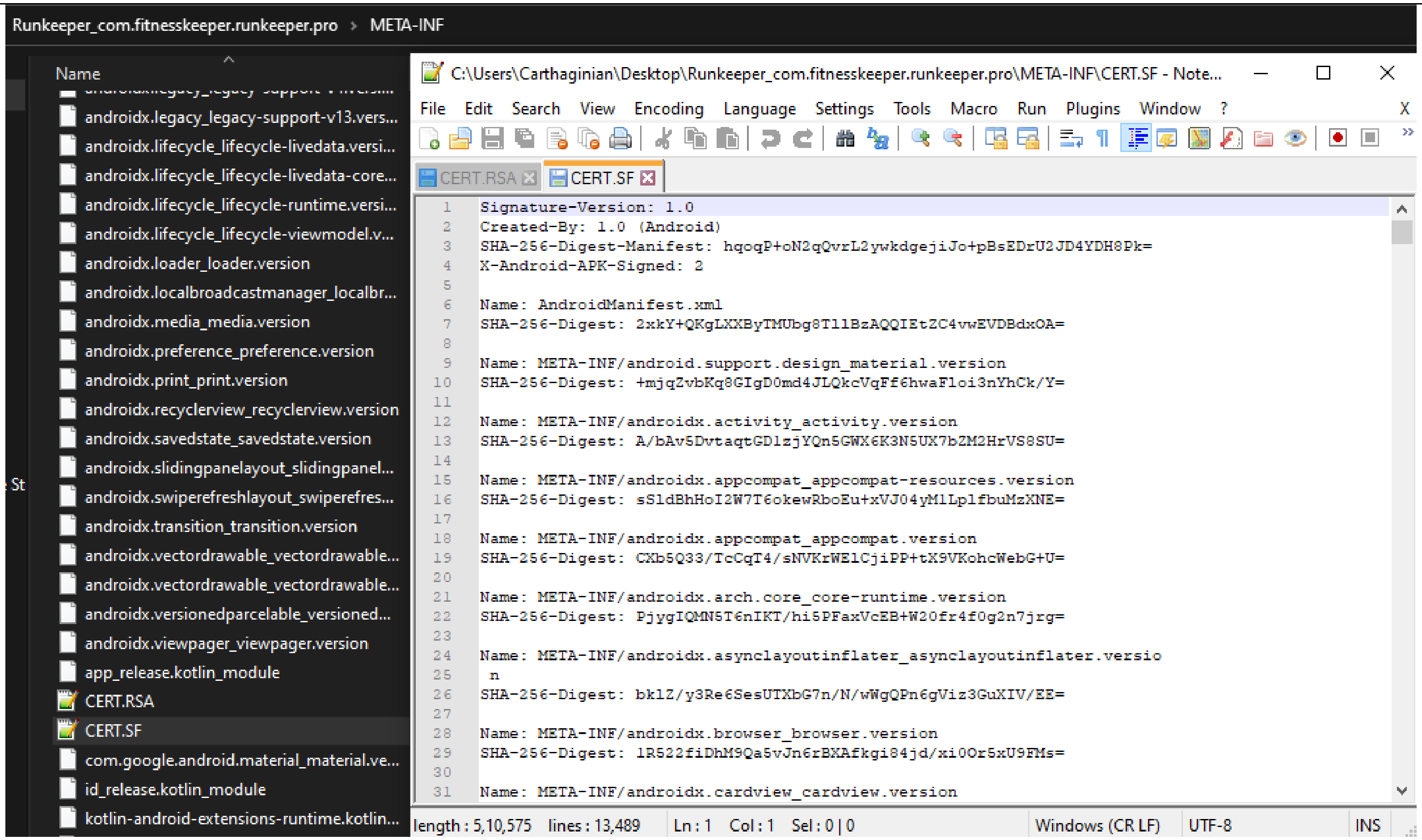
C:\Users\Carthaginian\Desktop\Runkeeper_com.fitnesskeeper.runkeeper.pro\META-INF\CERT.RSA - Not...

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
CERT.RSA

```
0,ETX2ACK   *†H†÷
SOHBELSTX ,ETX‡0,ETXUSSTXSOHSOH1SI0
ACK `†HSOHeETXEOTSTXSOHENQNUL0VTACK *†H†÷
SOHBELSOH ,STXETX0,SOHÿ0,SOHh ETXSTXSOHSTXSTXEOTKÇ‰f0
ACK *†H†÷
SOHSOHENQENQNUL0C1SYN0DC4ACKETXUEOT
DC3
FitnessKeeper1DC20DLEACKETXUEOTVTDC3 RunKeeper1NAK0DC3ACKETXUEOTETXDC3FFJason Jacobs0 ETB
100415214718ZCANSI210903222147182OC1SYN0DC4ACKETXUEOT
DC3
FitnessKeeper1DC20DLEACKETXUEOTVTDC3 RunKeeper1NAK0DC3ACKETXUEOTETXDC3FFJason Jacobs0Ÿ0
ACK *†H†÷
SOHSOHSOHENQNULETXNUL0%STXNULÅ†5OETX‡;USõnBEL1XáÇLMFF*ÁÍ(ÇVTóBELCAN`›fkSOË:FS¹icÆBS ?LGc
>*ËßSTXETXSOHNULSOH0
ACK *†H†÷
SOHSOHENQENQNULETXNULSO«ETBDLE;DC4ªÆNÃtÛûGø¿™›__9ŸRë†+Áõ1ÉMETXFFLó³}¬k¶SOì—=CtåVÁéDC2èßSY
DC3
FitnessKeeper1DC20DLEACKETXUEOTVTDC3 RunKeeper1NAK0DC3ACKETXUEOTETXDC3FFJason JacobsSTXEO
ACK `†HSOHeETXEOTSTXSOHENQNUL0
ACK *†H†÷
SOHSOHSOHENQNULEOT€8Óâ>Oc9Ü‡¾±  Üa7USáâ:uŰ~DC1+SI
¢Œñ¯Œç'–'i¥dvPÑ[.Ûhfz¼øÌß¡ç^~
À÷P¨ÑÜTèÙeDC4¢Mÿñã¿'qÆ^*ÓRSSUB4$vVTSUBæ‹ðCdØ4ª[°à ߎëETBDC2o®Ù+BS¥ÙàûV¼Â §¬^mú
```

length : 822   lines : 23   Ln : 1   Col : 1   Sel : 0 | 0   Macintosh (CR)   ANSI   INS

Source: Contents of Runkeeper - GPS Track Run Walk (https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro, Last accessed on Mar 19, 2020) as an example of ASICS Digital app


Runkeeper_com.fitnesskeeper.runkeeper.pro › META-INF
Name
androidx.legacy_legacy-support-v13.vers...
androidx.lifecycle_lifecycle-livedata.versi...
androidx.lifecycle_lifecycle-livedata-core...
androidx.lifecycle_lifecycle-runtime.versi...
androidx.lifecycle_lifecycle-viewmodel.v...
androidx.loader_loader.version
androidx.localbroadcastmanager_localbr...
androidx.media_media.version
androidx.preference_preference.version
androidx.print_print.version
androidx.recyclerview_recyclerview.version
androidx.savedstate_savedstate.version
androidx.slidingpanelayout_slidingpanel...
androidx.swiperefreshlayout_swiperefres...
androidx.transition_transition.version
androidx.vectordrawable_vectordrawable...
androidx.vectordrawable_vectordrawable...
androidx.versionedparcelable_versioned...
androidx.viewpager_viewpager.version
app_release.kotlin_module
CERT.RSA
CERT.SF
com.google.android.material_material.ve...
id_release.kotlin_module
kotlin-android-extensions-runtime.kotlin...
C:\Users\Carthaginian\Desktop\Runkeeper_com.fitnesskeeper.runkeeper.pro\META-INF\CERT.SF - Note...
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
CERT.RSA
CERT.SF
length : 5,10,575 lines : 13,489 Ln : 1 Col : 1 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

```
Signature-Version: 1.0
Created-By: 1.0 (Android)
SHA-256-Digest-Manifest: hqoqP+oN2qQvrL2ywkdgejiJo+pBsEDrU2JD4YDH8Pk=
X-Android-APK-Signed: 2

Name: AndroidManifest.xml
SHA-256-Digest: 2xkY+QKgLXXByTMUbg8T11BzAQQIEtZC4vwEVDBdxOA=

Name: META-INF/android.support.design_material.version
SHA-256-Digest: +mjqZvbKq8GIgD0md4JLQkcVqFf6hwaFloi3nYhCk/Y=

Name: META-INF/androidx.activity_activity.version
SHA-256-Digest: A/bAv5DvtaqtGDlzjYQn5GWX6K3N5UX7bZM2HrVS8SU=

Name: META-INF/androidx.appcompat_appcompat-resources.version
SHA-256-Digest: sSldBhHoI2W7T6okewRboEu+xVJ04yMlLplfbuMzXNE=

Name: META-INF/androidx.appcompat_appcompat.version
SHA-256-Digest: CXb5Q33/TcCqT4/sNVKrWElCjiPP+tX9VKohcWebG+U=

Name: META-INF/androidx.arch.core_core-runtime.version
SHA-256-Digest: PjygIQMN5T6nIKT/hi5PFaxVcEB+W20fr4f0g2n7jrg=

Name: META-INF/androidx.asynclayoutinflater_asynclayoutinflater.versio
 n
SHA-256-Digest: bklZ/y3Re6SesUTXbG7n/N/wWgQPn6gViz3GuXIV/EE=

Name: META-INF/androidx.browser_browser.version
SHA-256-Digest: 1R522fiDhM9Qa5vJn6rBXAfkgi84jd/xi0Or5xU9FMs=

Name: META-INF/androidx.cardview_cardview.version
```

Source: Contents of Runkeeper - GPS Track Run Walk (https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro, Last accessed on Mar 19, 2020) as an example of ASICS Digital app

By signing the app binary with a digital signature, ASICS Digital's mobile apps are tamper resistant enabling Google and the Android mobile devices to verify that the application is being distributed by trusted source (*i.e.* ASICS Digital) and that the application has not been modified by a third party, which can be verified by the corresponding public key generated as part of the pair. Thus the app binary is made resistant to modification by digital signing.

Accordingly ASICS Digital’s Android mobile apps establish SSL/TLS communications with ASICS Digital’s servers, which involve a SSL/TLS handshake procedure involving asymmetric key encryption. SSL/TLS ensures secure communication and renders the mobile app data further resistant to observation.

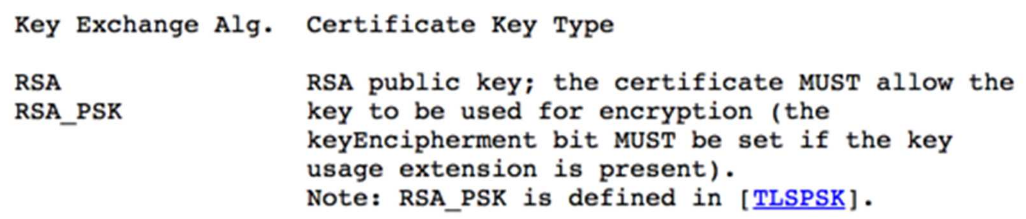
4. Resistant to Observation Because Mobile App is Stored on Remote System in Encrypted Form

The mobile app is made further resistant to observation because when downloaded and installed on a user’s Android mobile device, it is stored in an encrypted form. Android implements disk encryption for encrypting the operating system software, apps and all related data on a mobile device – which further renders ASICS Digital app resistant to observation[23].

5. Resistant to Observation Because Mobile App Securely Communicates with ASICS Digital Over SSL/TLS

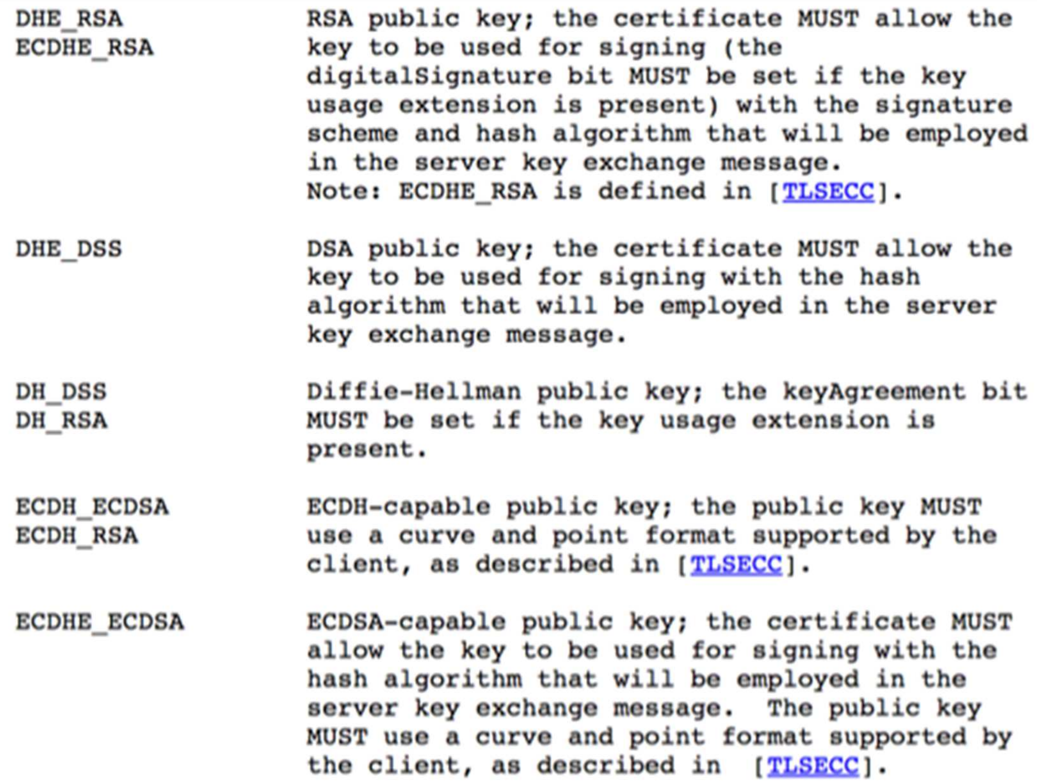
ASICS Digital’s mobile apps are made further resistant to observation, at least in part, because the mobile app communicates with ASICS Digital using SSL/TLS during operation. ASICS Digital app users establish SSL/TLS communications with ASICS Digital servers when the app is executed. Such secure communication is necessary to keep source code as well as user identity and activity from being observed in transit from the remote system to ASICS Digital servers and vice versa.

The secure communications process starts with a TLS handshake procedure which uses asymmetric key encryption and necessitates generation of at least one asymmetric key pair. Specifically, user’s remote device negotiates with ASICS Digital servers the cipher suite and the key exchange algorithm that will be used for the handshake.



```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].
```

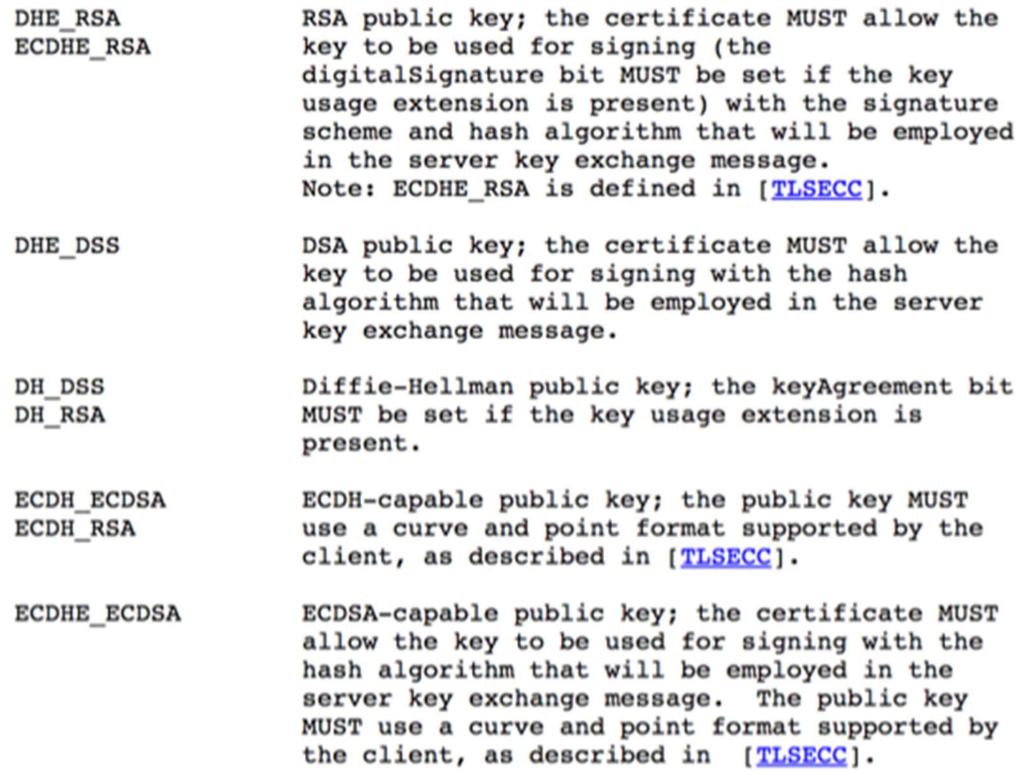
[23] *See, e.g.,* https://source.android.com/security/encryption/, Last accessed on Mar 19, 2020

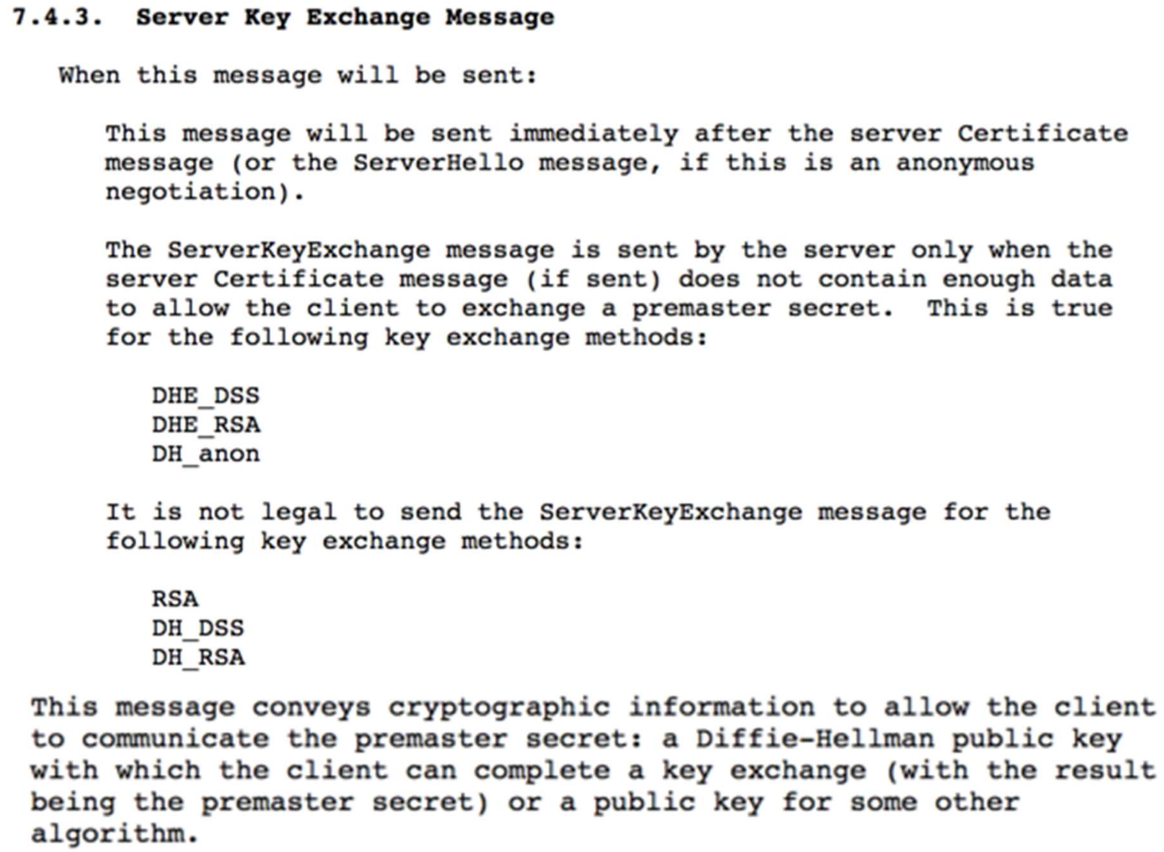
	 DHE_RSA RSA public key; the certificate MUST allow the ECDHE_RSA key to be used for signing (the digitalSignature bit MUST be set if the key usage extension is present) with the signature scheme and hash algorithm that will be employed in the server key exchange message. Note: ECDHE_RSA is defined in [TLSECC]. DHE_DSS DSA public key; the certificate MUST allow the key to be used for signing with the hash algorithm that will be employed in the server key exchange message. DH_DSS Diffie-Hellman public key; the keyAgreement bit DH_RSA MUST be set if the key usage extension is present. ECDH_ECDSA ECDH-capable public key; the public key MUST ECDH_RSA use a curve and point format supported by the client, as described in [TLSECC]. ECDHE_ECDSA ECDSA-capable public key; the certificate MUST allow the key to be used for signing with the hash algorithm that will be employed in the server key exchange message. The public key MUST use a curve and point format supported by the client, as described in [TLSECC]. *Source: https://tools.ietf.org/html/rfc5246 at 48-49*, Last accessed on Mar 19, 2020 Each of these algorithms necessitates generating one or more asymmetric key pairs – that are in turn used to compute a shared master secret for encrypting communication between ASICS Digital and user's remote device. For **RSA and RSA _PSK**, ASICS Digital server generates an RSA public-private key pair. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, ASICS Digital server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[24]. The user's remote device also generates a second Diffie-Hellman public-private key pair.

[24] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf,

	For **DHE_DSS and DH_DSS**, ASICS Digital server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user’s remote device also generates a second Diffie-Hellman public-private key pair. For **ECDH_ECDSA and ECDHE_ECDSA**, ASICS Digital server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user’s remote device also generates a second Diffie-Hellman public-private key pair.

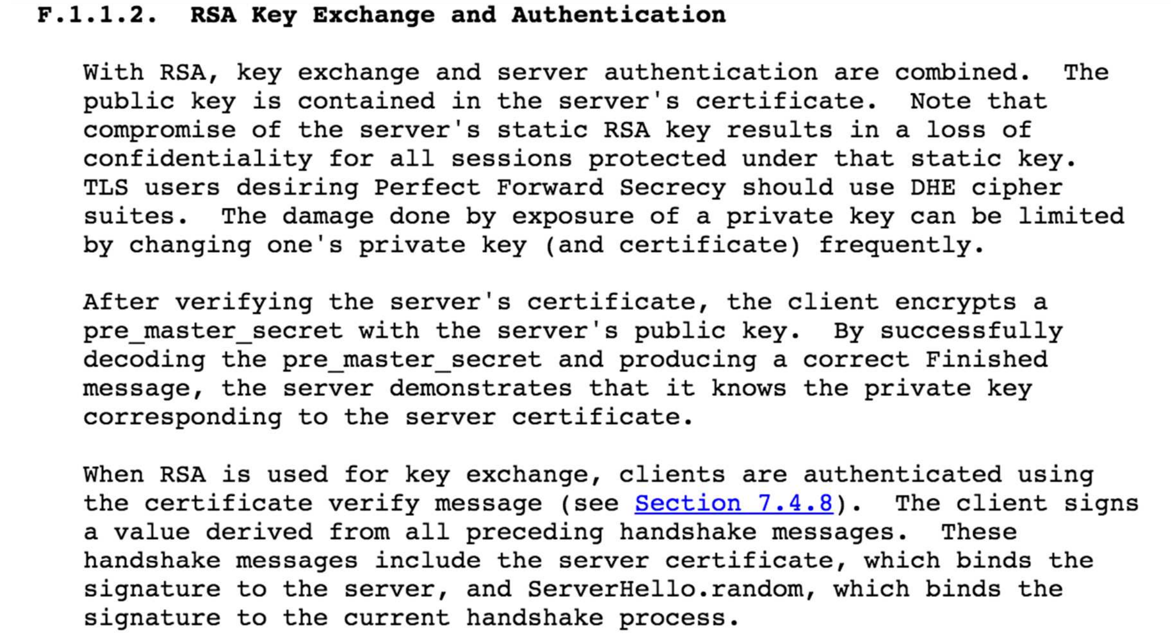
http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN
1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020

	 DHE_RSA RSA public key; the certificate MUST allow the ECDHE_RSA key to be used for signing (the digitalSignature bit MUST be set if the key usage extension is present) with the signature scheme and hash algorithm that will be employed in the server key exchange message. Note: ECDHE_RSA is defined in [TLSECC]. DHE_DSS DSA public key; the certificate MUST allow the key to be used for signing with the hash algorithm that will be employed in the server key exchange message. DH_DSS Diffie-Hellman public key; the keyAgreement bit DH_RSA MUST be set if the key usage extension is present. ECDH_ECDSA ECDH-capable public key; the public key MUST ECDH_RSA use a curve and point format supported by the client, as described in [TLSECC]. ECDHE_ECDSA ECDSA-capable public key; the certificate MUST allow the key to be used for signing with the hash algorithm that will be employed in the server key exchange message. The public key MUST use a curve and point format supported by the client, as described in [TLSECC]. *Source: https://tools.ietf.org/html/rfc5246 at 48-49*, Last accessed on Mar 19, 2020



```
7.4.3.  Server Key Exchange Message

   When this message will be sent:

      This message will be sent immediately after the server Certificate
      message (or the ServerHello message, if this is an anonymous
      negotiation).

      The ServerKeyExchange message is sent by the server only when the
      server Certificate message (if sent) does not contain enough data
      to allow the client to exchange a premaster secret.  This is true
      for the following key exchange methods:

         DHE_DSS
         DHE_RSA
         DH_anon

      It is not legal to send the ServerKeyExchange message for the
      following key exchange methods:

         RSA
         DH_DSS
         DH_RSA

  This message conveys cryptographic information to allow the client
  to communicate the premaster secret: a Diffie-Hellman public key
  with which the client can complete a key exchange (with the result
  being the premaster secret) or a public key for some other
  algorithm.
```

Source: https://tools.ietf.org/html/rfc5246 at 50-51, Last accessed on Mar 19, 2020

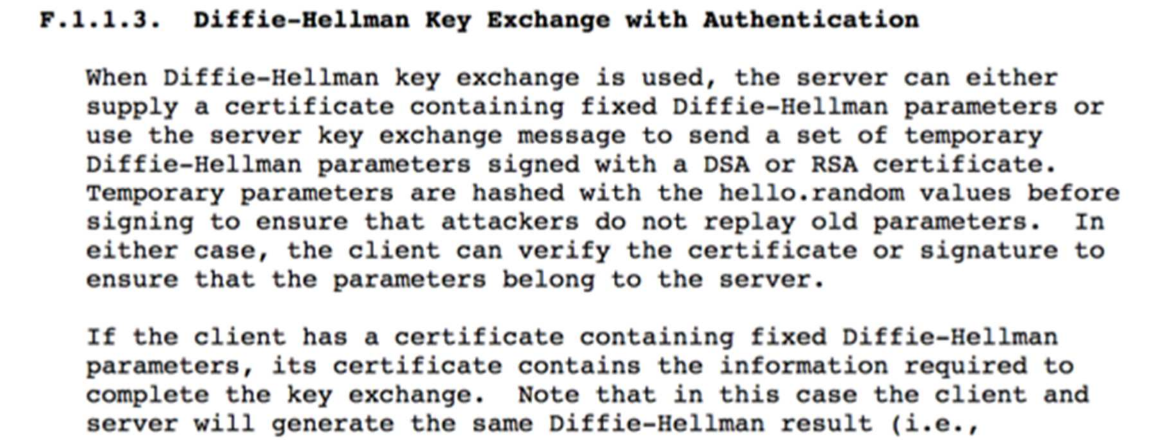


F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

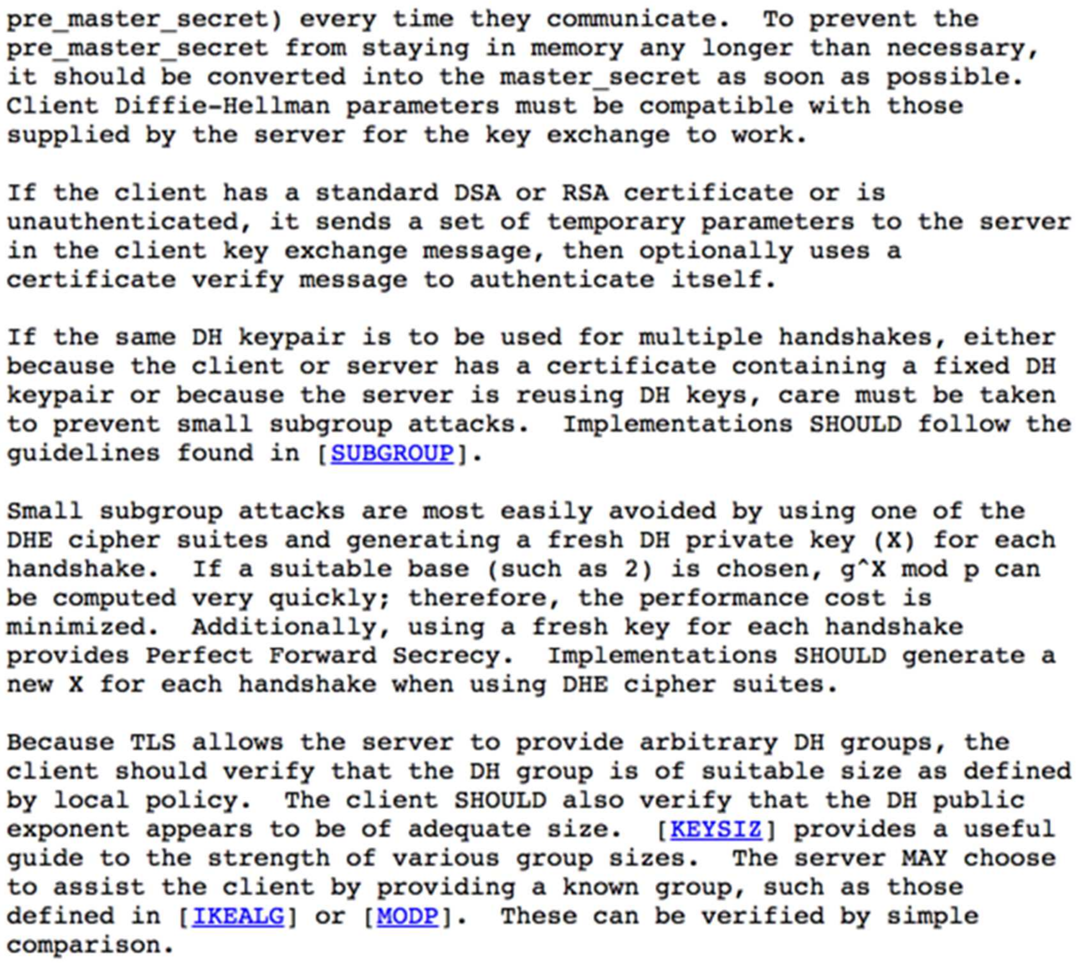
When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.



F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,

	 pre_master_secret) every time they communicate. To prevent the pre_master_secret from staying in memory any longer than necessary, it should be converted into the master_secret as soon as possible. Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work. If the client has a standard DSA or RSA certificate or is unauthenticated, it sends a set of temporary parameters to the server in the client key exchange message, then optionally uses a certificate verify message to authenticate itself. If the same DH keypair is to be used for multiple handshakes, either because the client or server has a certificate containing a fixed DH keypair or because the server is reusing DH keys, care must be taken to prevent small subgroup attacks. Implementations SHOULD follow the guidelines found in [SUBGROUP]. Small subgroup attacks are most easily avoided by using one of the DHE cipher suites and generating a fresh DH private key (X) for each handshake. If a suitable base (such as 2) is chosen, g^X mod p can be computed very quickly; therefore, the performance cost is minimized. Additionally, using a fresh key for each handshake provides Perfect Forward Secrecy. Implementations SHOULD generate a new X for each handshake when using DHE cipher suites. Because TLS allows the server to provide arbitrary DH groups, the client should verify that the DH group is of suitable size as defined by local policy. The client SHOULD also verify that the DH public exponent appears to be of adequate size. [KEYSIZ] provides a useful guide to the strength of various group sizes. The server MAY choose to assist the client by providing a known group, such as those defined in [IKEALG] or [MODP]. These can be verified by simple comparison. Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020 The generated asymmetric key pairs are then used to compute a shared master secret which is then used to encrypt subsequent communications between ASICS Digital and the user’s remote device so that they are resistant to observation during transit.

	For **RSA and RSA _PSK**, the RSA public-private key pair is used to encrypt a random premaster secret which is in turn used by ASICS Digital server and the user's remote device to compute a master secret. ASICS Digital and the user's remote device use the master secret for encrypting and decrypting communication messages. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, ASICS Digital server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[25]. The user's remote device also generates a second Diffie-Hellman public-private key pair. ASICS Digital server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Google's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret for encrypting and decrypting communication messages. For **DHE_DSS and DH_DSS**, ASICS Digital server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. ASICS Digital server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Google's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret for encrypting and decrypting communication messages. For **ECDH_ECDSA and ECDHE_ECDSA**, ASICS Digital server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. ASICS Digital server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Google's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret for encrypting and decrypting communication messages.
9. The method of claim 1, wherein building the executable tamper resistant code module comprises generating an integrity verification kernel.	ASICS Digital's mobile software application products and services including by way of example, but not limited to the following apps ("mobile applications", "mobile apps" or "Accused Products") that are specifically developed, used, sold, offered for sale, marketed, licensed and distributed by ASICS Digital to be downloaded onto Android mobile or tablet devices.

[25] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN 1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020

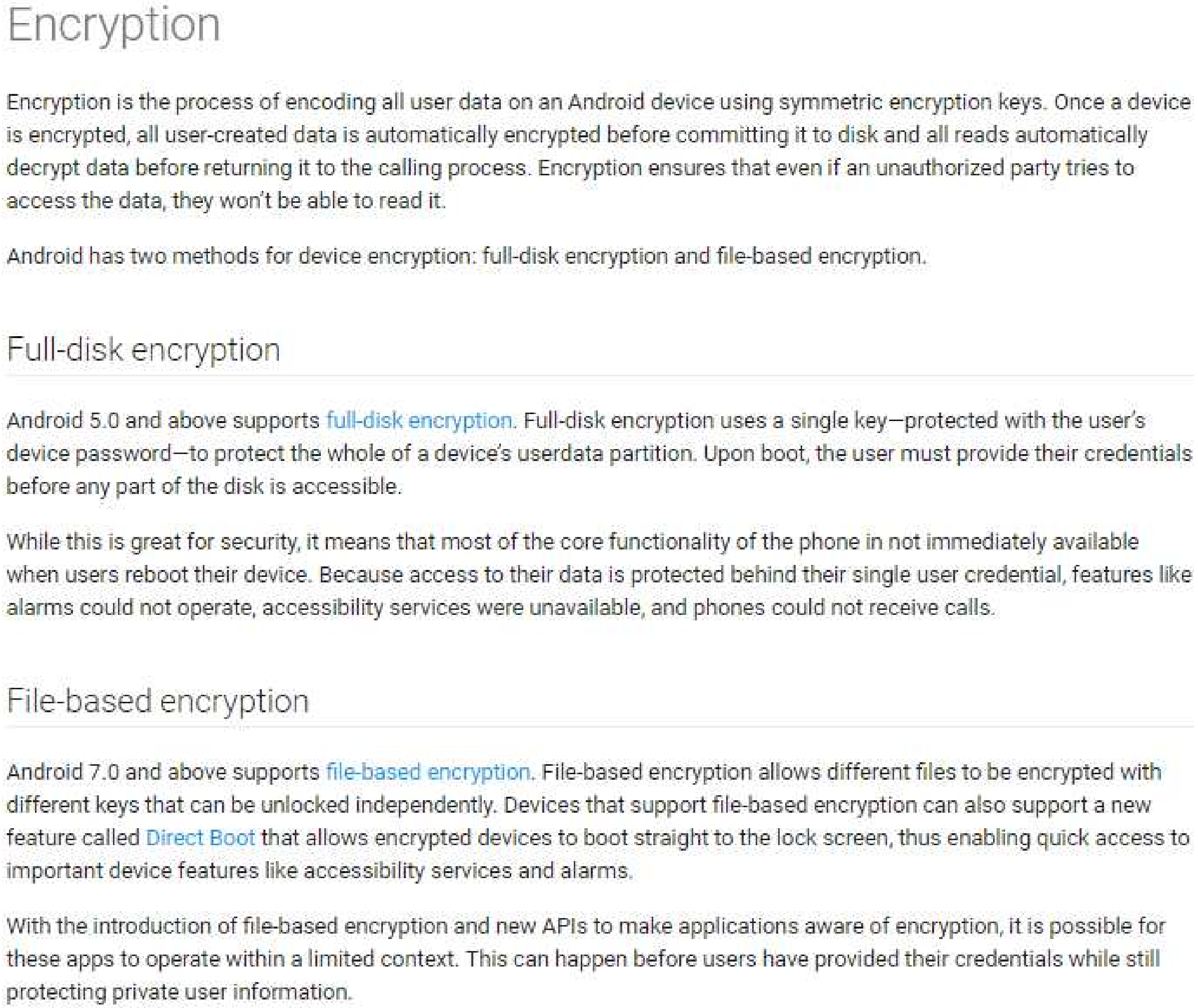
	• Runkeeper - GPS Track Run Walk (https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro, Last accessed on Mar 19, 2020) • ASICS Studio: Run, Lift & Get Fit (https://play.google.com/store/apps/details?id=com.fitnesskeeper.asicsstudio, Last accessed on Mar 19, 2020) ASICS Digital directly infringes and/or continues to knowingly induce Google to infringe this claim by intentionally developing, making, marketing, advertising, providing, sending, distributing and licensing its mobile applications software, documentation, materials, training or support and aiding, abetting, encouraging, promoting or inviting use thereof. Upon information and belief, the method step of generating an integrity verification kernel is performed by Google and/or its agents – whose acts are attributable to ASICS Digital (i) because ASICS Digital works together with Google in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Google distributes and markets ASICS Digital's mobile apps under the direction and control of ASICS Digital, or acts as agent, or on behalf of ASICS Digital, in the building, marketing and distribution of ASICS Digital's mobile apps. Alternatively, to the extent any portion of this method step is performed by ASICS Digital, such acts are attributable to Google, who conditions participation in and the receipt of a benefit, namely, the distribution of ASICS Digital's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Google in the building and upload of ASICS Digital's mobile apps, and ASICS Digital induces infringement by Google in the building, marketing and distribution of ASICS Digital's mobile apps. The Court construed "integrity verification kernel" to mean "software that verifies that a program image corresponds to a supplied digital signature and that is resistant to observation and modification." When a mobile app is deployed on device, a hash algorithm is used to compute hashes for the resources. The public key is used to decrypt the hashes in the hash manifest. Then, the hashes from the hash manifest are compared with the previous hash computations. If the hashes do not match, the digital signature is invalid and the application does not launch. Thus, the private key described above is used to digitally sign the program image for the app, and this digital signature is supplied so that the mobile app can be verified prior to launch of the mobile application[26]. The code for Android used to validate the digitally signed mobile app code is itself resistant to observation and modification. Further, Android implements disk encryption for encrypting the operating system software, applications and all related data on a mobile device – which renders the code for validating the digitally signed mobile app binary further resistant to observation[27].

[26] *See, e.g.,* https://developer.android.com/studio/publish/app-signing.html, Last accessed on Mar 19, 2020

[27] *See, e.g.,* https://source.android.com/security/encryption/, Last accessed on Mar 19, 2020

	Further Android implements a secure boot functionality that verifies the operating system, including code for validating digitally signed mobile apps, when the mobile device is powered on. If this verification fails, *i.e.* if the operating system has been maliciously modified, the operating system does not launch[28]. Thus, the software that verifies that a program image corresponds to a supplied digital signature is both resistant to observation and modification.

[28] *See, e.g.,* https://source.android.com/security/verifiedboot/ and http://www.zdnet.com/article/google-now-requires-full-device-encryption-on-new-android-6-0-devices/, Last accessed on Mar 19, 2020



Encryption

Encryption is the process of encoding all user data on an Android device using symmetric encryption keys. Once a device is encrypted, all user-created data is automatically encrypted before committing it to disk and all reads automatically decrypt data before returning it to the calling process. Encryption ensures that even if an unauthorized party tries to access the data, they won't be able to read it.

Android has two methods for device encryption: full-disk encryption and file-based encryption.

Full-disk encryption

Android 5.0 and above supports full-disk encryption. Full-disk encryption uses a single key—protected with the user's device password—to protect the whole of a device's userdata partition. Upon boot, the user must provide their credentials before any part of the disk is accessible.

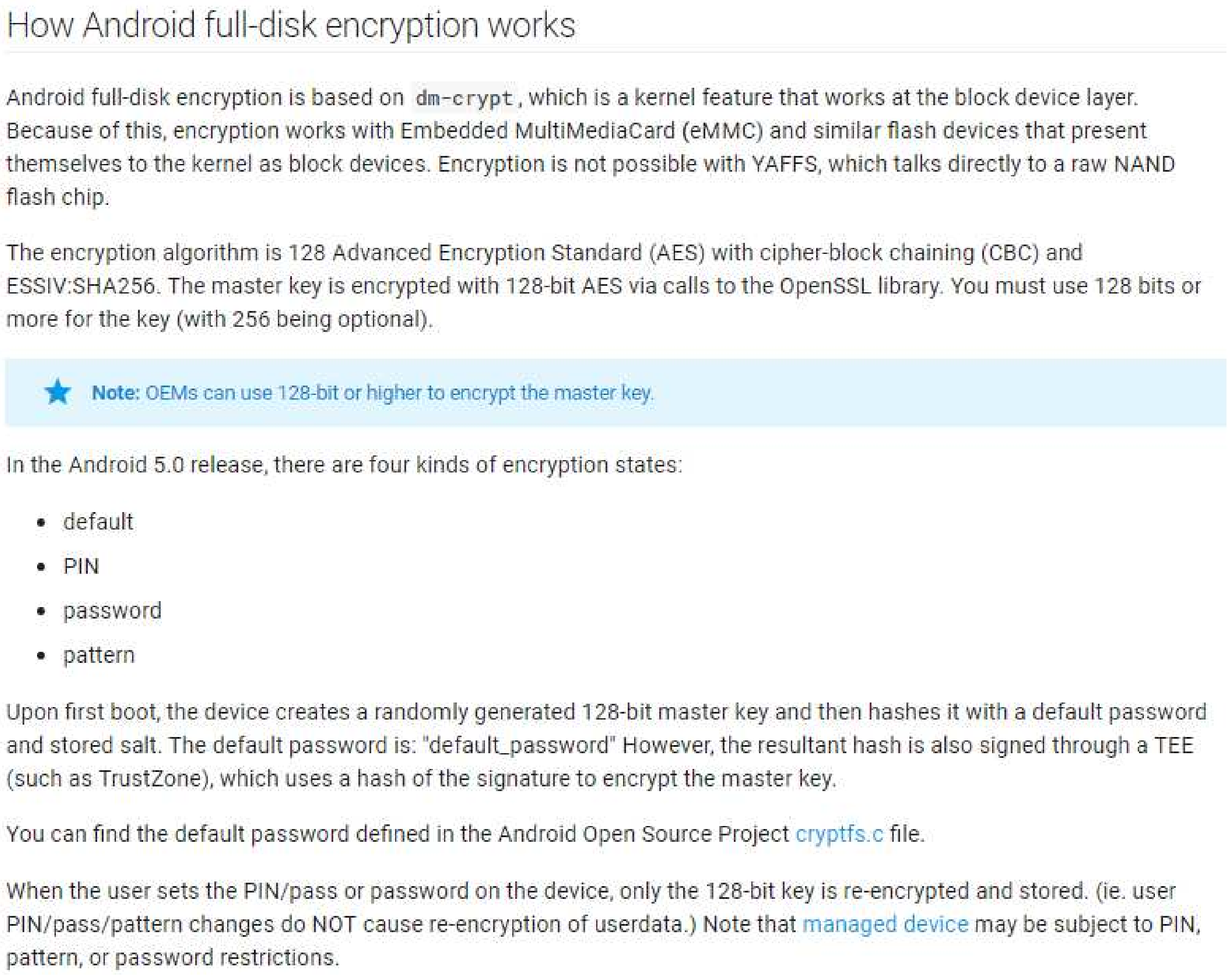
While this is great for security, it means that most of the core functionality of the phone in not immediately available when users reboot their device. Because access to their data is protected behind their single user credential, features like alarms could not operate, accessibility services were unavailable, and phones could not receive calls.

File-based encryption

Android 7.0 and above supports file-based encryption. File-based encryption allows different files to be encrypted with different keys that can be unlocked independently. Devices that support file-based encryption can also support a new feature called Direct Boot that allows encrypted devices to boot straight to the lock screen, thus enabling quick access to important device features like accessibility services and alarms.

With the introduction of file-based encryption and new APIs to make applications aware of encryption, it is possible for these apps to operate within a limited context. This can happen before users have provided their credentials while still protecting private user information.

Source: http://web.archive.org/web/20171208043438/https://source.android.com/security/encryption/, Last accessed on Mar 19, 2020



How Android full-disk encryption works

Android full-disk encryption is based on `dm-crypt`, which is a kernel feature that works at the block device layer. Because of this, encryption works with Embedded MultiMediaCard (eMMC) and similar flash devices that present themselves to the kernel as block devices. Encryption is not possible with YAFFS, which talks directly to a raw NAND flash chip.

The encryption algorithm is 128 Advanced Encryption Standard (AES) with cipher-block chaining (CBC) and ESSIV:SHA256. The master key is encrypted with 128-bit AES via calls to the OpenSSL library. You must use 128 bits or more for the key (with 256 being optional).

★ **Note:** OEMs can use 128-bit or higher to encrypt the master key.

In the Android 5.0 release, there are four kinds of encryption states:

- default
- PIN
- password
- pattern

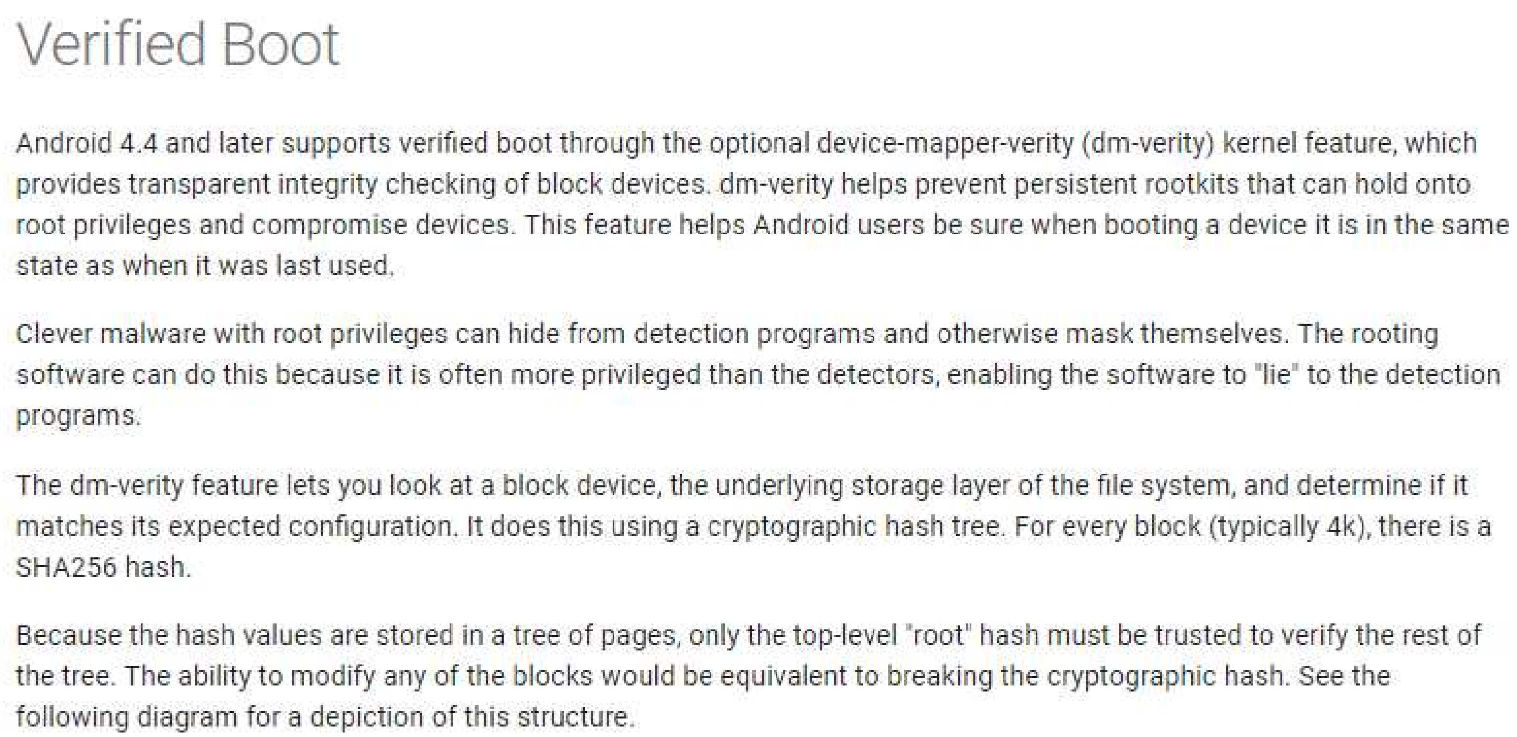
Upon first boot, the device creates a randomly generated 128-bit master key and then hashes it with a default password and stored salt. The default password is: "default_password" However, the resultant hash is also signed through a TEE (such as TrustZone), which uses a hash of the signature to encrypt the master key.

You can find the default password defined in the Android Open Source Project cryptfs.c file.

When the user sets the PIN/pass or password on the device, only the 128-bit key is re-encrypted and stored. (ie. user PIN/pass/pattern changes do NOT cause re-encryption of userdata.) Note that managed device may be subject to PIN, pattern, or password restrictions.

Source: http://web.archive.org/web/20170912153704/https://source.android.com/security/encryption/full-disk , Last accessed on Mar 19, 2020

Further Android implements a secure boot functionality that verifies the operating system, including code for validating digitally signed mobile apps, when the mobile device is powered on. If this verification fails, i.e. if the operating system has been maliciously modified, the operating system does not launch.



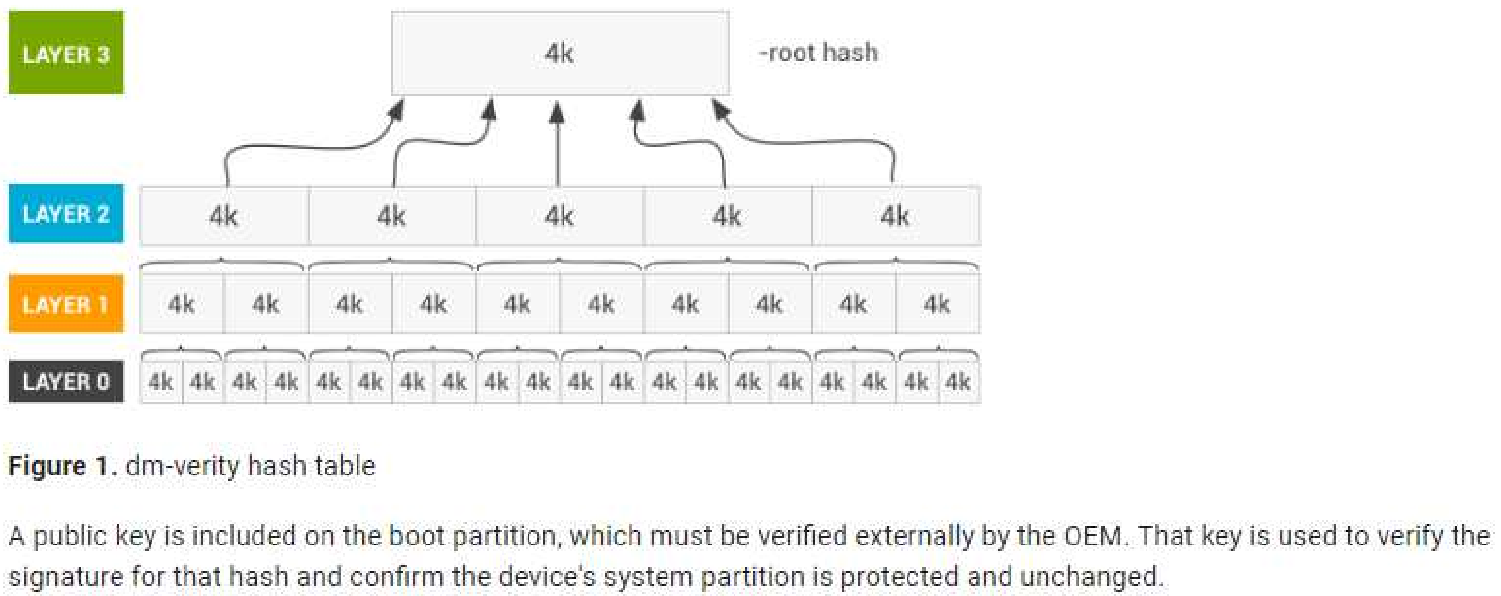
Verified Boot

Android 4.4 and later supports verified boot through the optional device-mapper-verity (dm-verity) kernel feature, which provides transparent integrity checking of block devices. dm-verity helps prevent persistent rootkits that can hold onto root privileges and compromise devices. This feature helps Android users be sure when booting a device it is in the same state as when it was last used.

Clever malware with root privileges can hide from detection programs and otherwise mask themselves. The rooting software can do this because it is often more privileged than the detectors, enabling the software to "lie" to the detection programs.

The dm-verity feature lets you look at a block device, the underlying storage layer of the file system, and determine if it matches its expected configuration. It does this using a cryptographic hash tree. For every block (typically 4k), there is a SHA256 hash.

Because the hash values are stored in a tree of pages, only the top-level "root" hash must be trusted to verify the rest of the tree. The ability to modify any of the blocks would be equivalent to breaking the cryptographic hash. See the following diagram for a depiction of this structure.

	 LAYER 3 4k -root hash LAYER 2 4k 4k 4k 4k 4k LAYER 1 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k LAYER 0 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k **Figure 1.** dm-verity hash table A public key is included on the boot partition, which must be verified externally by the OEM. That key is used to verify the signature for that hash and confirm the device's system partition is protected and unchanged. Source: http://web.archive.org/web/20171201215912/https://source.android.com/security/verifiedboot/[29], Last accessed on Mar 19, 2020 This verified boot became mandatory for Android 6.0 or later: https://source.android.com/compatibility/6.0/android-6.0-cdd, Last accessed on Mar 19, 2020
10. The method of claim 9, wherein generating an integrity verification kernel comprises accessing an asymmetric public key of a	ASICS Digital's mobile software application products and services including by way of example, but not limited to the following apps ("mobile applications", "mobile apps" or "Accused Products") that are specifically developed, used, sold, offered for sale, marketed, licensed and distributed by ASICS Digital to be downloaded onto Android mobile or tablet devices.

[29] See also http://www.tomshardware.com/news/marshmallow-encryption-fingerprints-verified-boot,30369.html, Last accessed on Mar 19, 2020

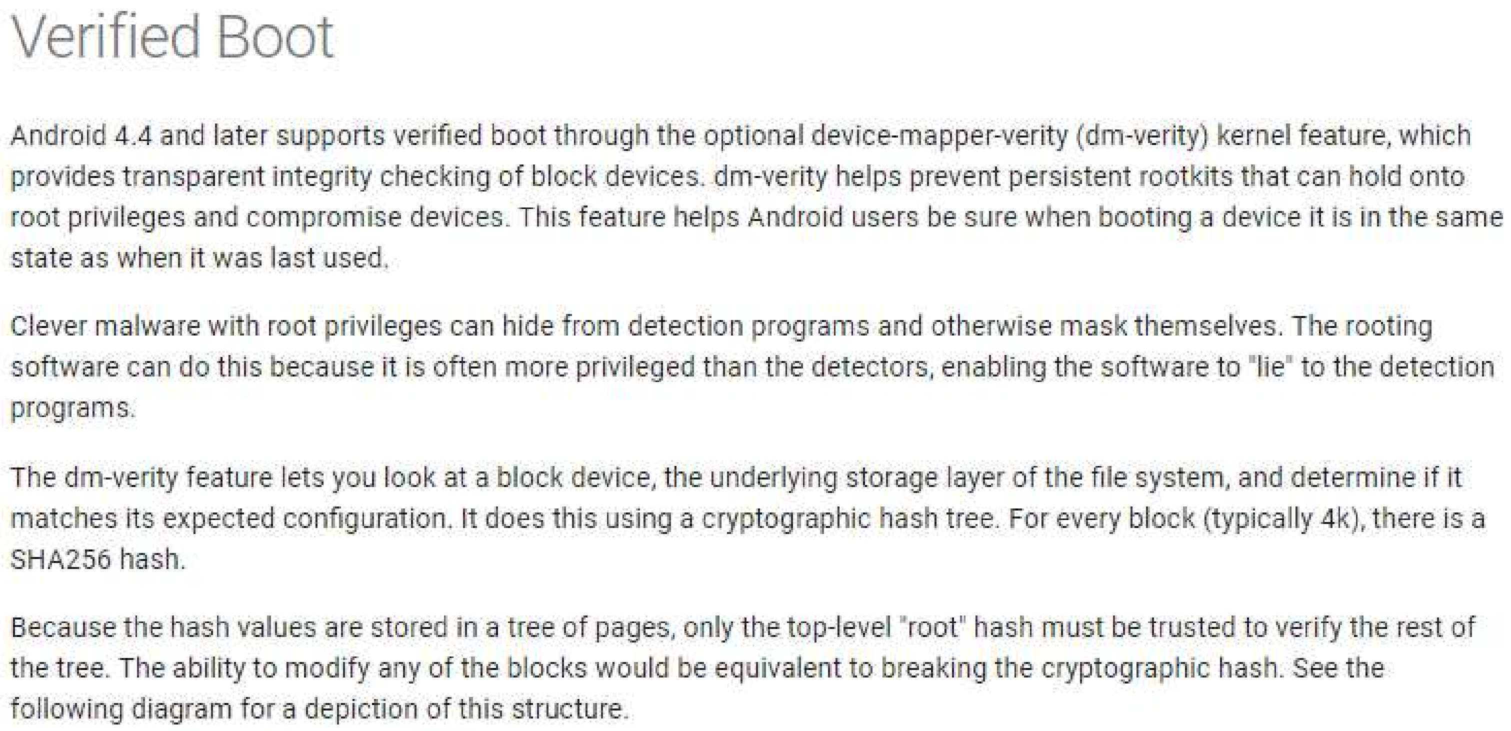
http://www.zdnet.com/article/google-now-requires-full-device-encryption-on-new-android-6-0-devices/, Last accessed on Mar 19, 2020

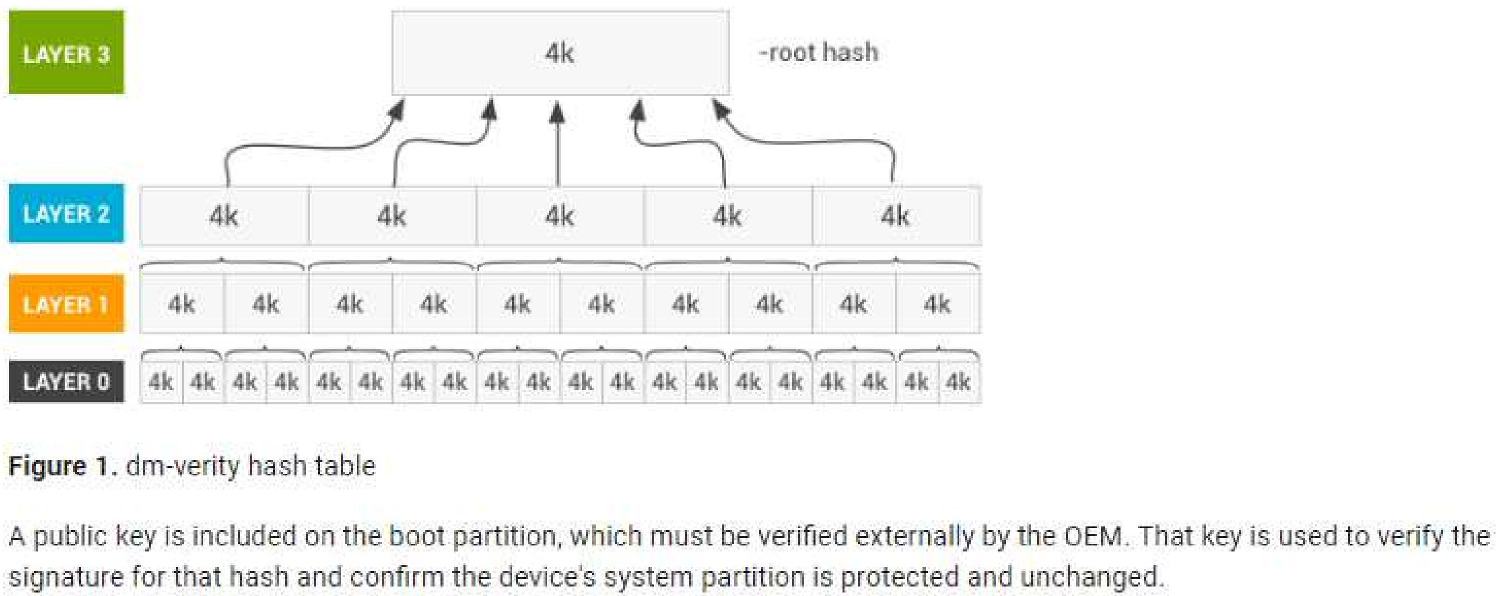
http://www.androidauthority.com/new-google-oem-marshmallow-requirements-650266/, Last accessed on Mar 19, 2020

https://www.linuxsecrets.com/elinux-wiki/images/b/b2/Android_Verified_Boot.pdf, Last accessed on Mar 19, 2020

predetermined asymmetric key pair associated with a manifest of the program signed by an asymmetric private key of the predetermined asymmetric key pair, producing integrity verification kernel code with the asymmetric public key for verifying the signed manifest of the program and combining manifest parser generator code and the integrity verification kernel code to produce the integrity verification kernel.	• Runkeeper - GPS Track Run Walk (https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro, Last accessed on Mar 19, 2020) • ASICS Studio: Run, Lift & Get Fit (https://play.google.com/store/apps/details?id=com.fitnesskeeper.asicsstudio, Last accessed on Mar 19, 2020) ASICS Digital directly infringes and/or continues to knowingly induce Google to infringe this claim by intentionally developing, making, marketing, advertising, providing, sending, distributing and licensing its mobile applications software, documentation, materials, training or support and aiding, abetting, encouraging, promoting or inviting use thereof. Upon information and belief, the method step of accessing an asymmetric public key of a predetermined asymmetric key pair associated with a manifest of the program signed by an asymmetric private key of the predetermined asymmetric key pair, producing integrity verification kernel code with the asymmetric public key for verifying the signed manifest of the program and combining manifest parser generator code and the integrity verification kernel code to produce the integrity verification kernel is performed by Google and/or its agents – whose acts are attributable to ASICS Digital (i) because ASICS Digital works together with Google in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Google distributes and markets ASICS Digital's mobile apps under the direction and control of ASICS Digital, or acts as agent, or on behalf of ASICS Digital, in the building, marketing and distribution of ASICS Digital's mobile apps. Alternatively, to the extent any portion of this method step is performed by ASICS Digital, such acts are attributable to Google, who conditions participation in and the receipt of a benefit, namely, the distribution of ASICS Digital's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Google in the building and upload of ASICS Digital's mobile apps, and ASICS Digital induces infringement by Google in the building, marketing and distribution of ASICS Digital's mobile apps. The Court previously construed[30] "integrity verification kernel" to mean "software that verifies that a program image corresponds to a supplied digital signature and that is resistant to observation and modification" and "manifest" to mean ""static source code that includes the integrity verification kernel's entry code, generator code, accumulator code, and other code for tamper detection." Android implements a secure boot functionality that verifies the operating system, including code for validating digitally signed mobile apps, when the mobile device is powered on. If this verification fails, i.e. if the operating system has been maliciously modified, the operating system, along with the mobile apps installed on the device, does not launch. Thus every time the device is restarted an integrity verification kernel is generated by accessing an asymmetric public key of a predetermined asymmetric key pair associated with a manifest of the operating system signed with the corresponding asymmetric private key by Google. This integrity verification kernel is implemented in code which is produced by combining code that

[30] See Memorandum Opinion and Order, Document 104 signed by Judge Rodney Gilstrap on 7/21/2016 in re Plano Encryption Technologies, LLC v. American Bank of Texas (2:15-cv-01273).

	parses a manifest associated with the operating system and code that verifies that the operating system on the device matches with the manifest and has not been maliciously modified. Further, the software that performs the above verification is stored in a compiled and encrypted form and is hence resistant to observation. Additionally, the software is digitally signed by Google with their asymmetric private key and is thus resistant to modification.  Verified Boot Android 4.4 and later supports verified boot through the optional device-mapper-verity (dm-verity) kernel feature, which provides transparent integrity checking of block devices. dm-verity helps prevent persistent rootkits that can hold onto root privileges and compromise devices. This feature helps Android users be sure when booting a device it is in the same state as when it was last used. Clever malware with root privileges can hide from detection programs and otherwise mask themselves. The rooting software can do this because it is often more privileged than the detectors, enabling the software to "lie" to the detection programs. The dm-verity feature lets you look at a block device, the underlying storage layer of the file system, and determine if it matches its expected configuration. It does this using a cryptographic hash tree. For every block (typically 4k), there is a SHA256 hash. Because the hash values are stored in a tree of pages, only the top-level "root" hash must be trusted to verify the rest of the tree. The ability to modify any of the blocks would be equivalent to breaking the cryptographic hash. See the following diagram for a depiction of this structure.

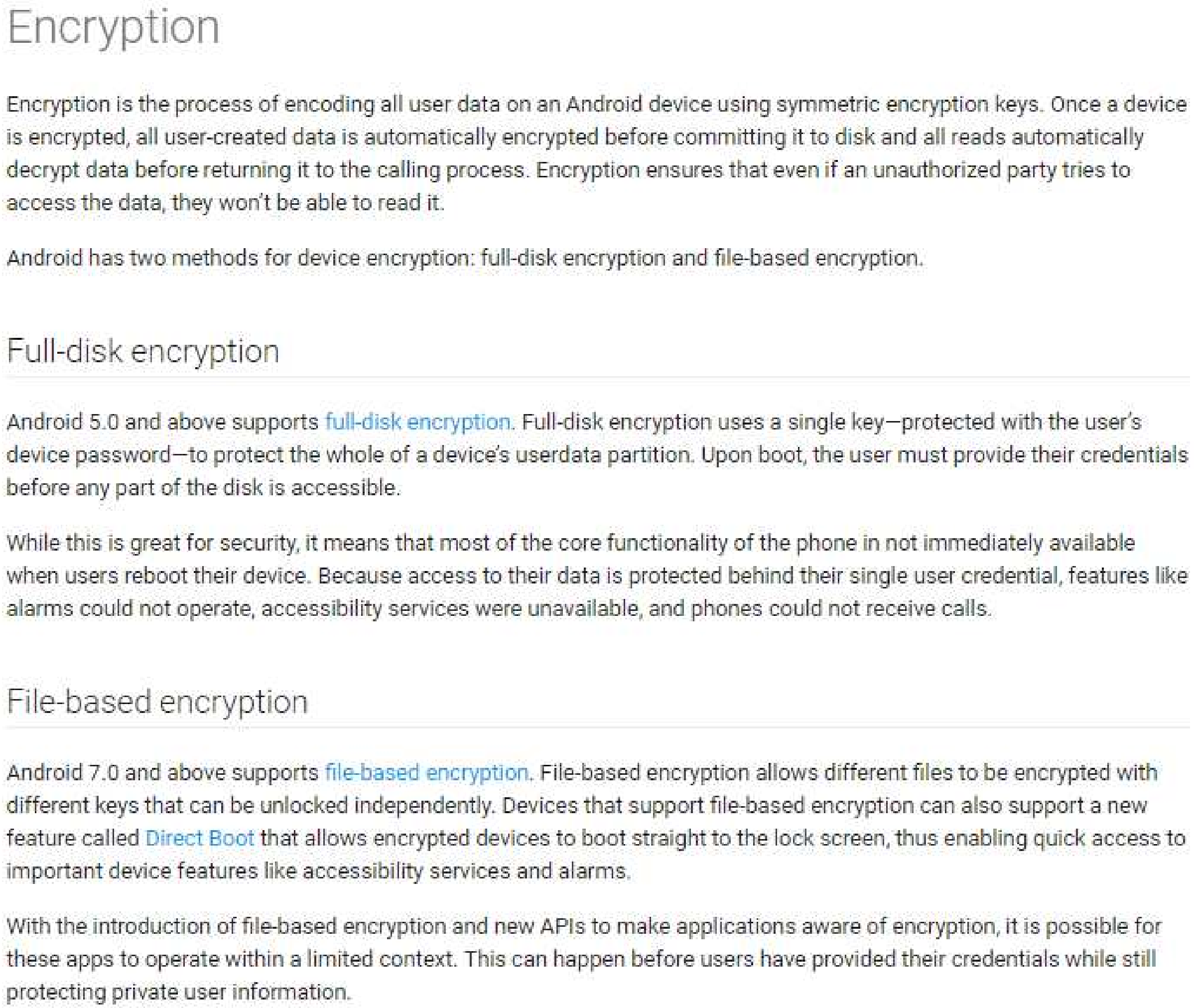
	 LAYER 3 4k -root hash LAYER 2 4k 4k 4k 4k 4k LAYER 1 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k LAYER 0 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k **Figure 1.** dm-verity hash table A public key is included on the boot partition, which must be verified externally by the OEM. That key is used to verify the signature for that hash and confirm the device's system partition is protected and unchanged. Source: http://web.archive.org/web/20171201215912/https://source.android.com/security/verifiedboot/[31], Last accessed on Mar 19, 2020 This verified boot became mandatory for Android 6.0 or later: https://source.android.com/compatibility/6.0/android-6.0-cdd, Last accessed on Mar 19, 2020

[31] See also http://www.tomshardware.com/news/marshmallow-encryption-fingerprints-verified-boot,30369.html, Last accessed on Mar 19, 2020

http://www.zdnet.com/article/google-now-requires-full-device-encryption-on-new-android-6-0-devices/, Last accessed on Mar 19, 2020

http://www.androidauthority.com/new-google-oem-marshmallow-requirements-650266/, Last accessed on Mar 19, 2020

https://www.linuxsecrets.com/elinux-wiki/images/b/b2/Android_Verified_Boot.pdf, Last accessed on Mar 19, 2020



Encryption

Encryption is the process of encoding all user data on an Android device using symmetric encryption keys. Once a device is encrypted, all user-created data is automatically encrypted before committing it to disk and all reads automatically decrypt data before returning it to the calling process. Encryption ensures that even if an unauthorized party tries to access the data, they won't be able to read it.

Android has two methods for device encryption: full-disk encryption and file-based encryption.

Full-disk encryption

Android 5.0 and above supports full-disk encryption. Full-disk encryption uses a single key—protected with the user's device password—to protect the whole of a device's userdata partition. Upon boot, the user must provide their credentials before any part of the disk is accessible.

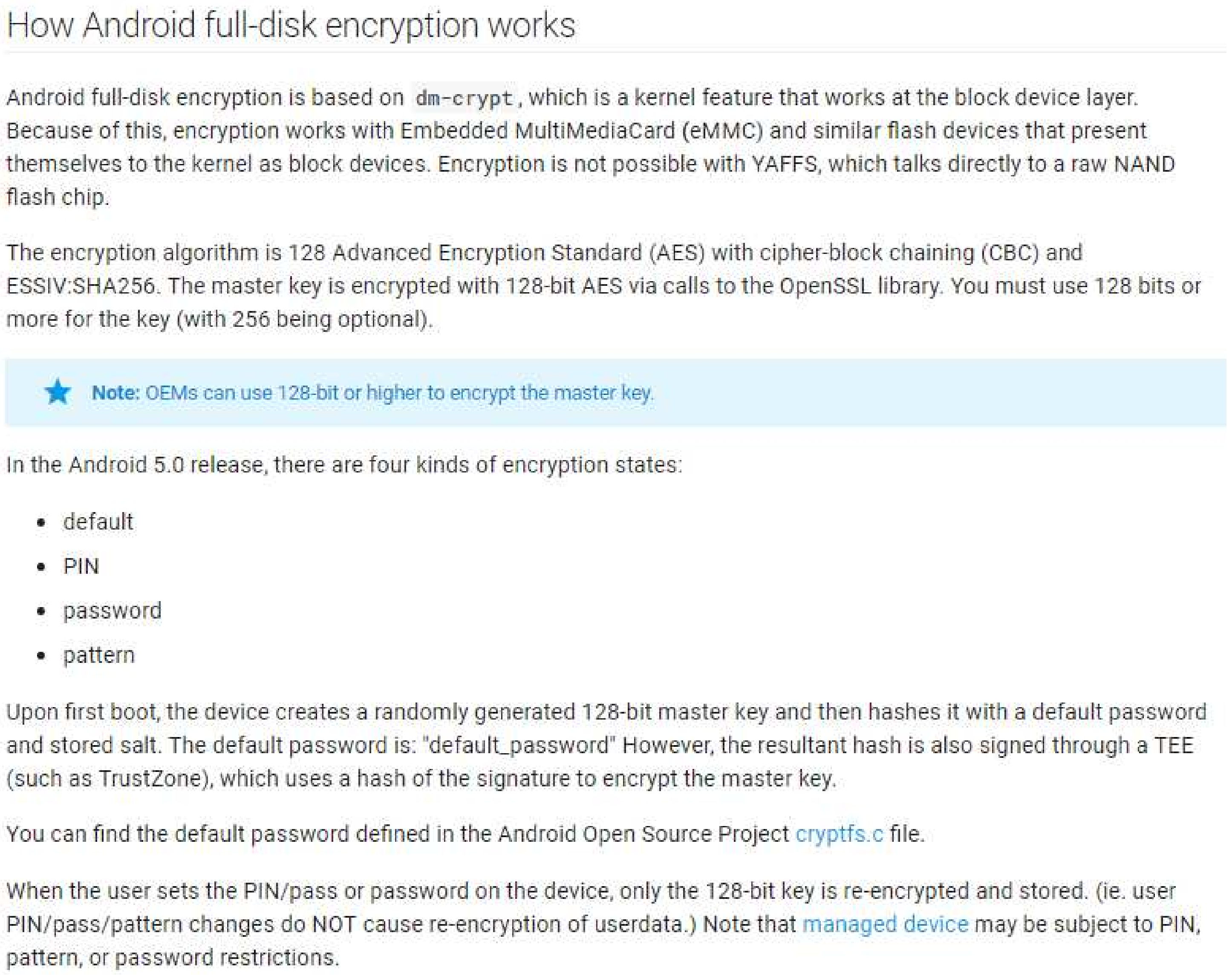
While this is great for security, it means that most of the core functionality of the phone in not immediately available when users reboot their device. Because access to their data is protected behind their single user credential, features like alarms could not operate, accessibility services were unavailable, and phones could not receive calls.

File-based encryption

Android 7.0 and above supports file-based encryption. File-based encryption allows different files to be encrypted with different keys that can be unlocked independently. Devices that support file-based encryption can also support a new feature called Direct Boot that allows encrypted devices to boot straight to the lock screen, thus enabling quick access to important device features like accessibility services and alarms.

With the introduction of file-based encryption and new APIs to make applications aware of encryption, it is possible for these apps to operate within a limited context. This can happen before users have provided their credentials while still protecting private user information.

Source: http://web.archive.org/web/20171208043438/https://source.android.com/security/encryption/, Last accessed on Mar 19, 2020



How Android full-disk encryption works

Android full-disk encryption is based on `dm-crypt`, which is a kernel feature that works at the block device layer. Because of this, encryption works with Embedded MultiMediaCard (eMMC) and similar flash devices that present themselves to the kernel as block devices. Encryption is not possible with YAFFS, which talks directly to a raw NAND flash chip.

The encryption algorithm is 128 Advanced Encryption Standard (AES) with cipher-block chaining (CBC) and ESSIV:SHA256. The master key is encrypted with 128-bit AES via calls to the OpenSSL library. You must use 128 bits or more for the key (with 256 being optional).

★ **Note:** OEMs can use 128-bit or higher to encrypt the master key.

In the Android 5.0 release, there are four kinds of encryption states:

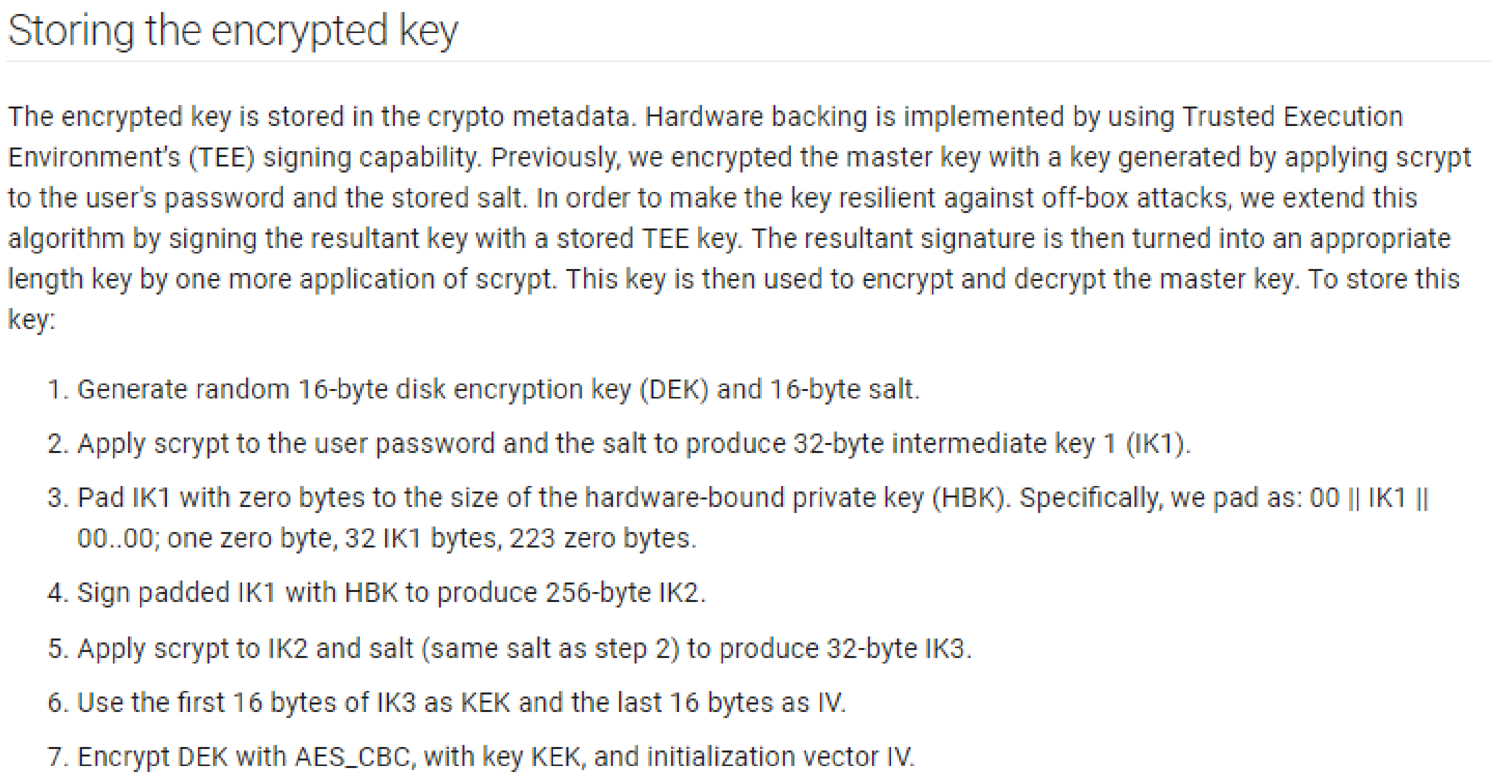
- default
- PIN
- password
- pattern

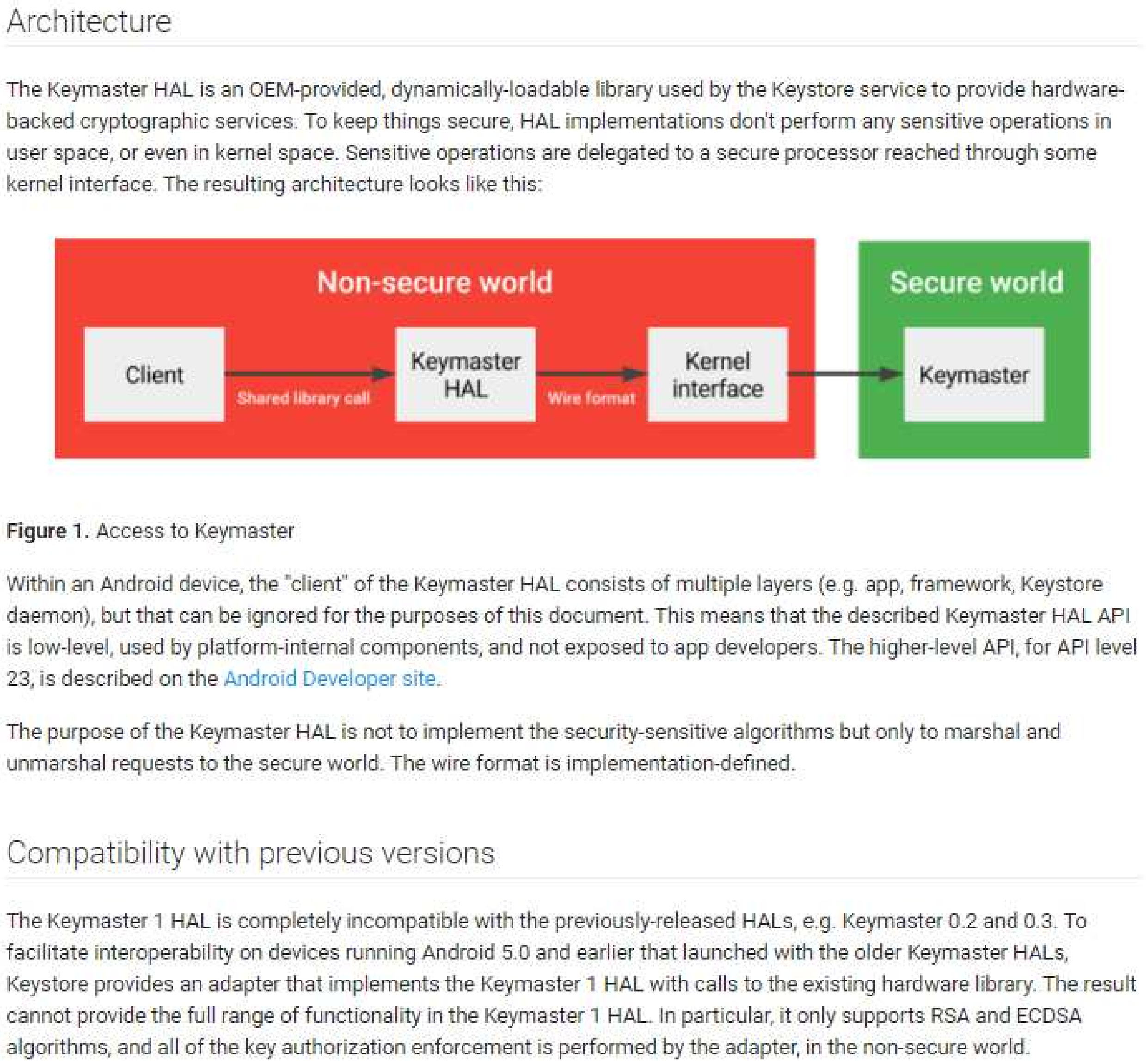
Upon first boot, the device creates a randomly generated 128-bit master key and then hashes it with a default password and stored salt. The default password is: "default_password" However, the resultant hash is also signed through a TEE (such as TrustZone), which uses a hash of the signature to encrypt the master key.

You can find the default password defined in the Android Open Source Project cryptfs.c file.

When the user sets the PIN/pass or password on the device, only the 128-bit key is re-encrypted and stored. (ie. user PIN/pass/pattern changes do NOT cause re-encryption of userdata.) Note that managed device may be subject to PIN, pattern, or password restrictions.

Source: http://web.archive.org/web/20170912153704/https://source.android.com/security/encryption/full-disk, Last accessed on Mar 19, 2020

	 **Storing the encrypted key** The encrypted key is stored in the crypto metadata. Hardware backing is implemented by using Trusted Execution Environment's (TEE) signing capability. Previously, we encrypted the master key with a key generated by applying scrypt to the user's password and the stored salt. In order to make the key resilient against off-box attacks, we extend this algorithm by signing the resultant key with a stored TEE key. The resultant signature is then turned into an appropriate length key by one more application of scrypt. This key is then used to encrypt and decrypt the master key. To store this key: 1. Generate random 16-byte disk encryption key (DEK) and 16-byte salt. 2. Apply scrypt to the user password and the salt to produce 32-byte intermediate key 1 (IK1). 3. Pad IK1 with zero bytes to the size of the hardware-bound private key (HBK). Specifically, we pad as: 00 || IK1 || 00..00; one zero byte, 32 IK1 bytes, 223 zero bytes. 4. Sign padded IK1 with HBK to produce 256-byte IK2. 5. Apply scrypt to IK2 and salt (same salt as step 2) to produce 32-byte IK3. 6. Use the first 16 bytes of IK3 as KEK and the last 16 bytes as IV. 7. Encrypt DEK with AES_CBC, with key KEK, and initialization vector IV. Source: https://web.archive.org/web/20171203224317/https://source.android.com/security/encryption/full-disk, Last accessed on Mar 19, 2020



Architecture

The Keymaster HAL is an OEM-provided, dynamically-loadable library used by the Keystore service to provide hardware-backed cryptographic services. To keep things secure, HAL implementations don't perform any sensitive operations in user space, or even in kernel space. Sensitive operations are delegated to a secure processor reached through some kernel interface. The resulting architecture looks like this:

Figure 1. Access to Keymaster

Within an Android device, the "client" of the Keymaster HAL consists of multiple layers (e.g. app, framework, Keystore daemon), but that can be ignored for the purposes of this document. This means that the described Keymaster HAL API is low-level, used by platform-internal components, and not exposed to app developers. The higher-level API, for API level 23, is described on the Android Developer site.

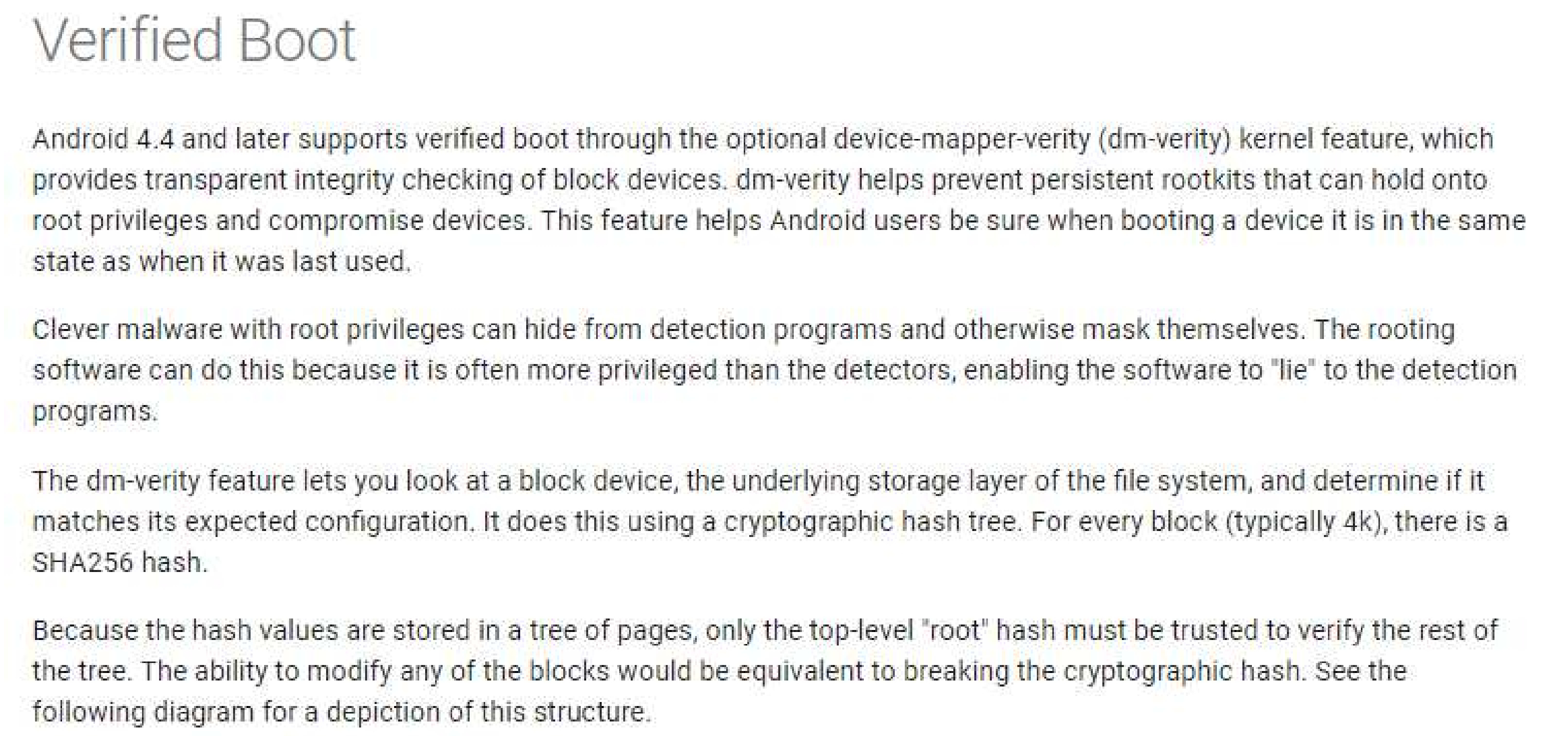
The purpose of the Keymaster HAL is not to implement the security-sensitive algorithms but only to marshal and unmarshal requests to the secure world. The wire format is implementation-defined.

Compatibility with previous versions

The Keymaster 1 HAL is completely incompatible with the previously-released HALs, e.g. Keymaster 0.2 and 0.3. To facilitate interoperability on devices running Android 5.0 and earlier that launched with the older Keymaster HALs, Keystore provides an adapter that implements the Keymaster 1 HAL with calls to the existing hardware library. The result cannot provide the full range of functionality in the Keymaster 1 HAL. In particular, it only supports RSA and ECDSA algorithms, and all of the key authorization enforcement is performed by the adapter, in the non-secure world.

Source: https://web.archive.org/web/20170912190232/https://source.android.com/security/keystore/, Last accessed on Mar 19, 2020

11. The method of claim 10, wherein the program comprises a trusted player and the method further comprises building a manifest for the trusted player, signing the manifest with the asymmetric private key of the predetermined asymmetric key pair, and storing the asymmetric public key of the predetermined asymmetric key pair.	ASICS Digital's mobile software application products and services including by way of example, but not limited to the following apps ("mobile applications", "mobile apps" or "Accused Products") that are specifically developed, used, sold, offered for sale, marketed, licensed and distributed by ASICS Digital to be downloaded onto Android mobile or tablet devices. • Runkeeper - GPS Track Run Walk (https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro, Last accessed on Mar 19, 2020) • ASICS Studio: Run, Lift & Get Fit (https://play.google.com/store/apps/details?id=com.fitnesskeeper.asicsstudio, Last accessed on Mar 19, 2020) ASICS Digital directly infringes and/or continues to knowingly induce Google to infringe this claim by intentionally developing, making, marketing, advertising, providing, sending, distributing and licensing its mobile applications software, documentation, materials, training or support and aiding, abetting, encouraging, promoting or inviting use thereof. Upon information and belief, the method step of building a manifest for the trusted player, signing the manifest with the asymmetric private key of the predetermined asymmetric key pair, and storing the asymmetric public key of the predetermined asymmetric key pair is performed by Google and/or its agents – whose acts are attributable to ASICS Digital (i) because ASICS Digital works together with Google in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Google distributes and markets ASICS Digital's mobile apps under the direction and control of ASICS Digital, or acts as agent, or on behalf of ASICS Digital, in the building, marketing and distribution of ASICS Digital's mobile apps. Alternatively, to the extent any portion of this method step is performed by ASICS Digital, such acts are attributable to Google, who conditions participation in and the receipt of a benefit, namely, the distribution of ASICS Digital's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Google in the building and upload of ASICS Digital's mobile apps, and ASICS Digital induces infringement by Google in the building, marketing and distribution of ASICS Digital's mobile apps. As explained in claim 10, Android implements a secure boot functionality that verifies the operating system, including code for validating digitally signed mobile apps, when the mobile device is powered on. If this verification fails, i.e. if the operating system has been maliciously modified, the operating system, along with the mobile apps installed on the device, does not launch. When Google installs the bootloader code and the operating system on a device before the device is sold to a user, Google builds a manifest for the operating system (Google being the trusted player) and sign the manifest with their asymmetric private key and storing the corresponding asymmetric public key in the bootloader so that the manifest of the operating system can be verified during device restart.



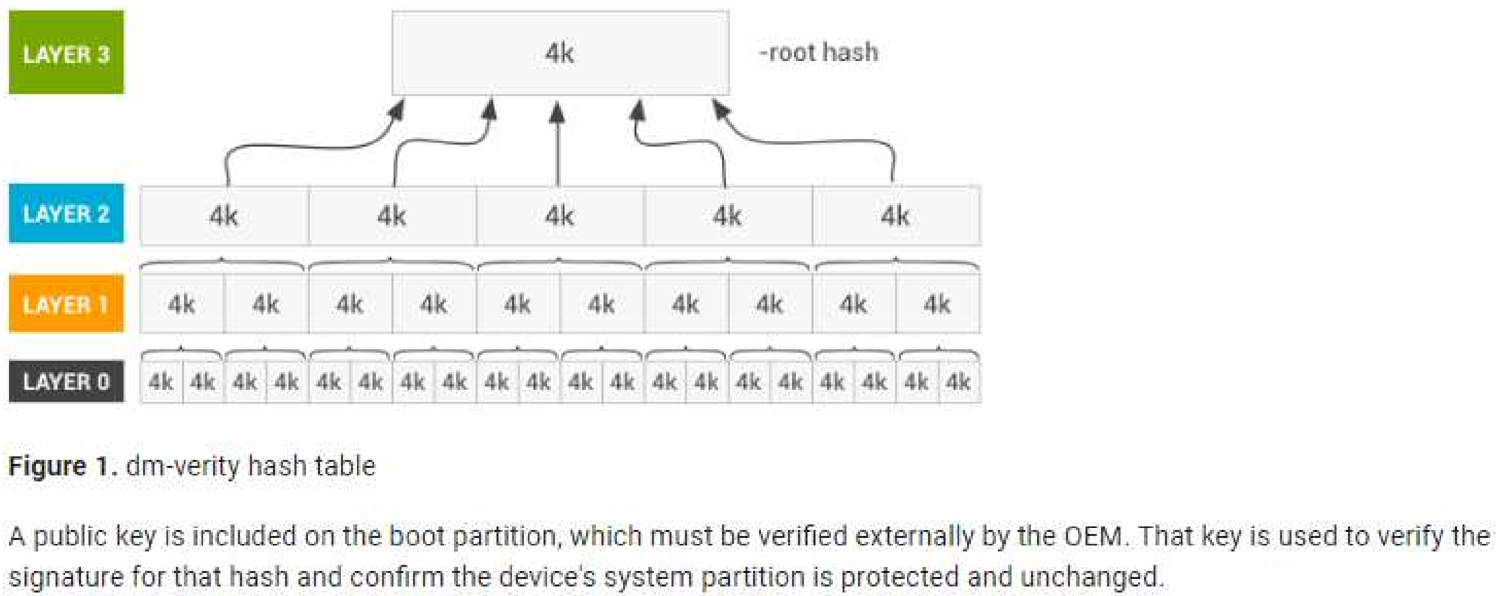
Verified Boot

Android 4.4 and later supports verified boot through the optional device-mapper-verity (dm-verity) kernel feature, which provides transparent integrity checking of block devices. dm-verity helps prevent persistent rootkits that can hold onto root privileges and compromise devices. This feature helps Android users be sure when booting a device it is in the same state as when it was last used.

Clever malware with root privileges can hide from detection programs and otherwise mask themselves. The rooting software can do this because it is often more privileged than the detectors, enabling the software to "lie" to the detection programs.

The dm-verity feature lets you look at a block device, the underlying storage layer of the file system, and determine if it matches its expected configuration. It does this using a cryptographic hash tree. For every block (typically 4k), there is a SHA256 hash.

Because the hash values are stored in a tree of pages, only the top-level "root" hash must be trusted to verify the rest of the tree. The ability to modify any of the blocks would be equivalent to breaking the cryptographic hash. See the following diagram for a depiction of this structure.

	 LAYER 3 4k -root hash LAYER 2 4k 4k 4k 4k 4k LAYER 1 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k LAYER 0 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k **Figure 1.** dm-verity hash table A public key is included on the boot partition, which must be verified externally by the OEM. That key is used to verify the signature for that hash and confirm the device's system partition is protected and unchanged. Source: http://web.archive.org/web/20171201215912/https://source.android.com/security/verifiedboot/[32], Last accessed on Mar 19, 2020 This verified boot became mandatory for Android 6.0 or later: https://source.android.com/compatibility/6.0/android-6.0-cdd, Last accessed on Mar 19, 2020
34. A method of securely distributing data encrypted by a public key of an asymmetric key pair comprising:	ASICS Digital's mobile software application products and services including by way of example, but not limited to the following apps ("mobile applications", "mobile apps" or "Accused Products") that are specifically developed, used, sold, offered for sale, marketed, licensed and distributed by ASICS Digital to be downloaded onto Android mobile or tablet devices.

[32] See also http://www.tomshardware.com/news/marshmallow-encryption-fingerprints-verified-boot,30369.html, Last accessed on Mar 19, 2020

http://www.zdnet.com/article/google-now-requires-full-device-encryption-on-new-android-6-0-devices/, Last accessed on Mar 19, 2020

http://www.androidauthority.com/new-google-oem-marshmallow-requirements-650266/, Last accessed on Mar 19, 2020

https://www.linuxsecrets.com/elinux-wiki/images/b/b2/Android_Verified_Boot.pdf, Last accessed on Mar 19, 2020

	- Runkeeper - GPS Track Run Walk (https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro, Last accessed on Mar 19, 2020) - ASICS Studio: Run, Lift & Get Fit (https://play.google.com/store/apps/details?id=com.fitnesskeeper.asicsstudio, Last accessed on Mar 19, 2020) ASICS Digital directly infringes and/or continues to knowingly induce Google to infringe this claim by intentionally developing, making, marketing, advertising, providing, sending, distributing and licensing its mobile applications software, documentation, materials, training or support and aiding, abetting, encouraging, promoting or inviting use thereof. To the extent any steps identified herein are performed by Google, such acts are attributable to ASICS Digital (i) because ASICS Digital works together with Google in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Google distributes and markets ASICS Digital's mobile apps under the direction and control of ASICS Digital, or acts as agent, or on behalf of ASICS Digital, in the building, marketing and distribution of ASICS Digital's mobile apps. Alternatively, any steps or acts performed by ASICS Digital, are attributable to Google, who conditions participation in and the receipt of a benefit, namely, the distribution of ASICS Digital's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Google in the building and upload of ASICS Digital's mobile apps, and ASICS Digital induces infringement by Google in the building, marketing and distribution of ASICS Digital's mobile apps. To the extent the preamble is limiting, ASICS Digital distributes data according to the method of claim 34 as set forth below. In order to build and send the mobile app securely to the Google servers, ASICS Digital practices the method of claim 34 as set forth below in order to securely distribute its mobile app to ASICS Digital's customers through the Google Play Store. Android Developer Console (https://play.google.com/apps/publish/, Last accessed on Mar 19, 2020) establishes SSL/TLS communications when uploading ASICS Digital's apps, as evidenced by the "https" in the URL. That process necessarily uses the public key of the generated asymmetric key pair to encrypt data that is used to create a symmetric key for secure communications.
building an executable tamper resistant key module identified for a selected program resident on a remote system, the executable tamper	Relevant to this claim element is the Court's previous construction[33] of "executable tamper resistant key module" / "executable tamper resistant code module" / "tamper resistant key module" to mean "software that is designed to work with other software, that is resistant to observation and

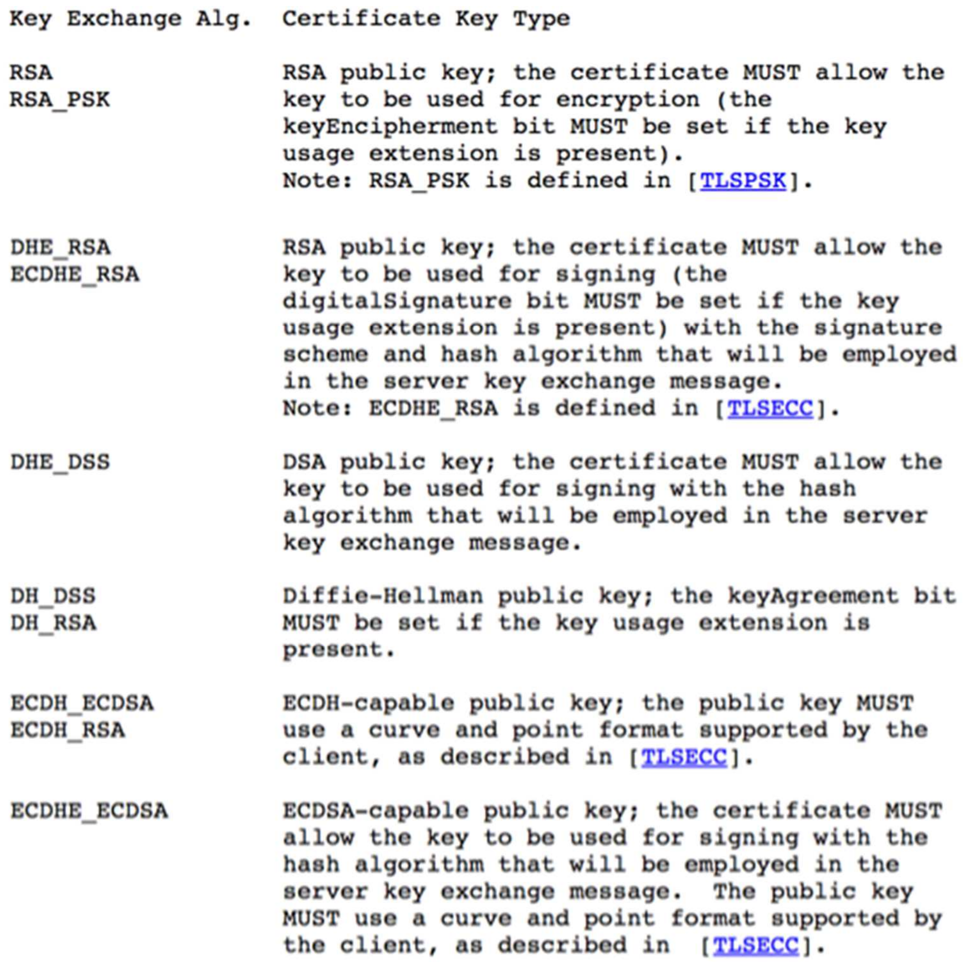
[33] See Memorandum Opinion and Order, Document 104 signed by Judge Rodney Gilstrap on 7/21/2016 in re Plano Encryption Technologies, LLC v. American Bank of Texas (2:15-cv-01273).

resistant key module including a private key of the asymmetric key pair and the encrypted data; and	modification, and that includes a key for secure communication." Also relevant to this claim element is the Court's previous rejection of limiting "including" to compiling[34]. The method step of "building an executable tamper resistant key module identified for a selected program resident on a remote system, the executable tamper resistant key module including a private key of the asymmetric key pair and the encrypted data" is performed by ASICS Digital and/or its agents. To the extent any portion of the method step is performed by Google and/or its agents, such acts are attributable to ASICS Digital (i) because ASICS Digital works together with Google in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Google distributes and markets ASICS Digital's mobile apps under the direction and control of ASICS Digital, or acts as agent, or on behalf of ASICS Digital, in the building, marketing and distribution of ASICS Digital's mobile apps. Building of an "executable tamper resistant code module" (that is, the mobile app) requires the inclusion of at least the following different asymmetric key pairs: (1) An asymmetric key pair must be included in order to send the mobile app from ASICS Digital to Google securely by SSL/TLS; and (2) An asymmetric key pair must be included by ASICS Digital in order to digitally sign the mobile app with a private asymmetric key and to verify the mobile app has not been changed with the public key for Android compatible mobile apps. The asymmetric key pair that is included to digitally sign the mobile app is different from and in addition to the claimed asymmetric key pair used to securely upload the mobile app to the Google servers for distribution on the Google Play Store. Thus, at least one asymmetric key pair is included having the claimed public key and claimed private key in building the tamper resistant app so that the mobile app code can be sent uploaded to the Google servers using SSL/TLS protocol. This asymmetric key pair(s) (as with the asymmetric key pair used to digitally sign the app) is complementary as described below by performing complementary functions, such as encrypting and decrypting data and/or creating and verifying signatures. As described in greater detail below, the claimed public and private key are generated and used to securely upload the mobile app onto the Google servers by SSL/TLS when building the app, where the app includes the generated private key of the claimed asymmetric key pair(s) and the encrypted data.

[34] Although the exact language construed from the previous claims is not at issue in claim 34, also relevant is the Court's construction of "an asymmetric key pair having a public key and a private key" in claims 1, 9 and 10 to mean "one or more asymmetric key pairs, one of the asymmetric key pairs having the claimed public key and claimed private key, the asymmetric keys of an asymmetric key pair being complementary by performing complementary functions, such as encrypting and decrypting data or creating and verifying signatures." While not necessarily adopting the Court's construction, Honeyman has assumed that the public key and private key in claim 34 must perform complementary functions.

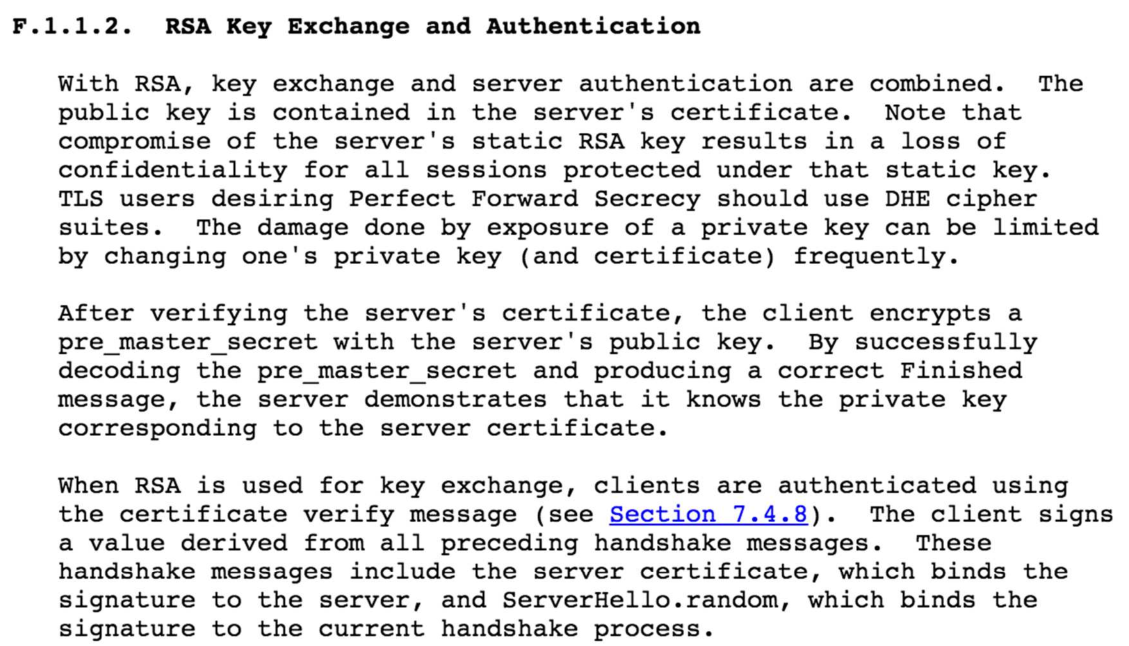
ASICS Digital uploads their mobile apps to Google using a TLS connection – which begins with a TLS handshake. A TLS handshake is a mandatory procedure that allows ASICS Digital and Google to exchange cryptographic parameters, including a cipher suite and arrive at a shared master secret for encrypting communication including upload of ASICS Digital's mobile apps to Google servers.

A TLS handshake begins with ASICS Digital sending a list of cipher suites supported by ASICS Digital to Google. These cipher suites specify at least one or more of the following key exchange algorithms:



```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].

DHE_RSA            RSA public key; the certificate MUST allow the
ECDHE_RSA          key to be used for signing (the
                   digitalSignature bit MUST be set if the key
                   usage extension is present) with the signature
                   scheme and hash algorithm that will be employed
                   in the server key exchange message.
                   Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS            DSA public key; the certificate MUST allow the
                   key to be used for signing with the hash
                   algorithm that will be employed in the server
                   key exchange message.

DH_DSS             Diffie-Hellman public key; the keyAgreement bit
DH_RSA             MUST be set if the key usage extension is
                   present.

ECDH_ECDSA         ECDH-capable public key; the public key MUST
ECDH_RSA           use a curve and point format supported by the
                   client, as described in [TLSECC].

ECDHE_ECDSA        ECDSA-capable public key; the certificate MUST
                   allow the key to be used for signing with the
                   hash algorithm that will be employed in the
                   server key exchange message.  The public key
                   MUST use a curve and point format supported by
                   the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

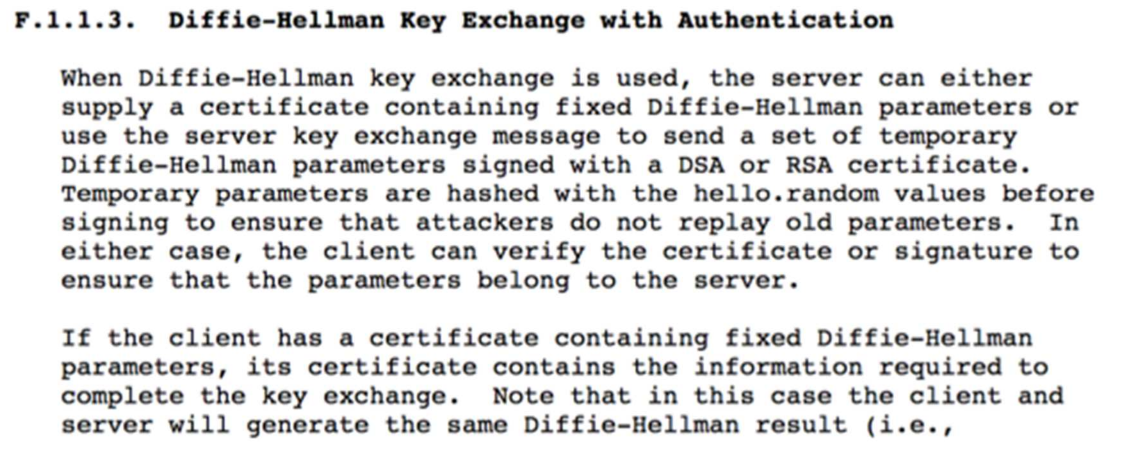


F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

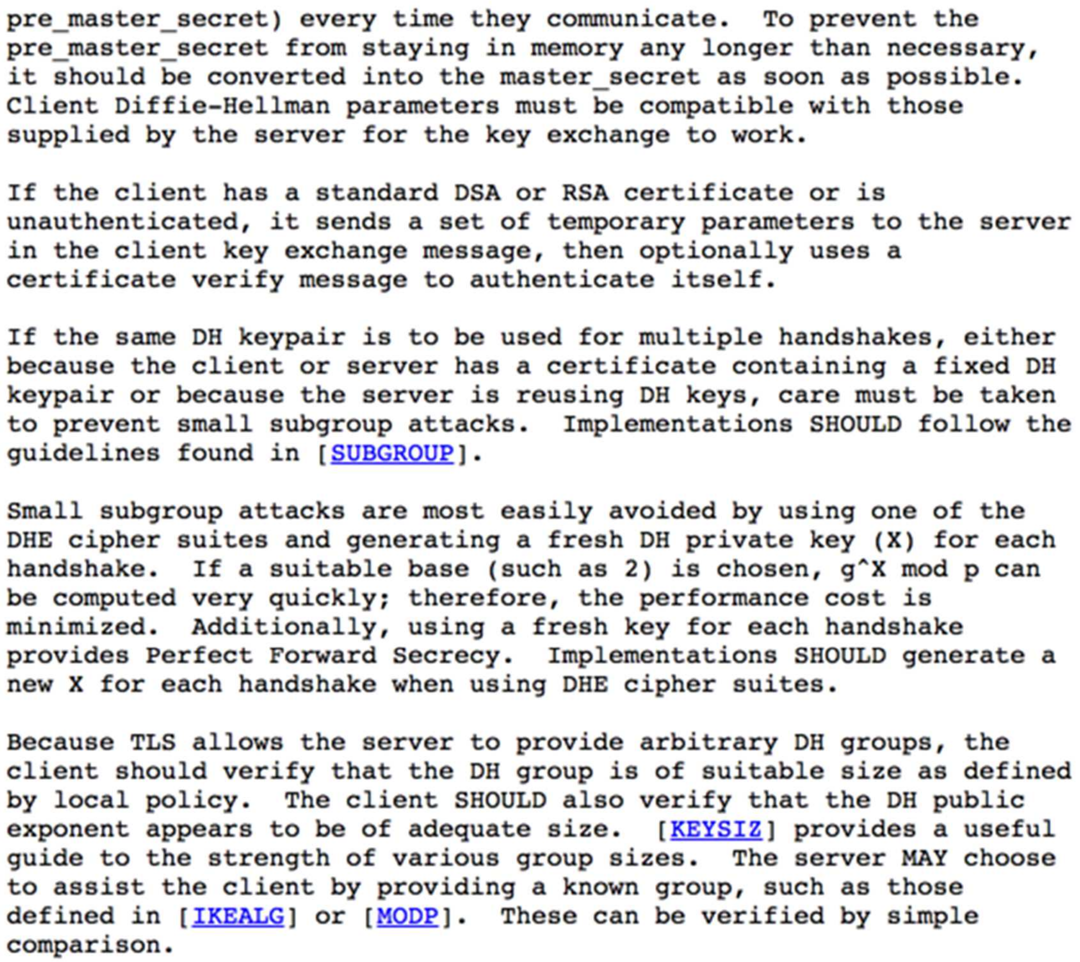
When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.



F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,

	 pre_master_secret) every time they communicate. To prevent the pre_master_secret from staying in memory any longer than necessary, it should be converted into the master_secret as soon as possible. Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work. If the client has a standard DSA or RSA certificate or is unauthenticated, it sends a set of temporary parameters to the server in the client key exchange message, then optionally uses a certificate verify message to authenticate itself. If the same DH keypair is to be used for multiple handshakes, either because the client or server has a certificate containing a fixed DH keypair or because the server is reusing DH keys, care must be taken to prevent small subgroup attacks. Implementations SHOULD follow the guidelines found in [SUBGROUP]. Small subgroup attacks are most easily avoided by using one of the DHE cipher suites and generating a fresh DH private key (X) for each handshake. If a suitable base (such as 2) is chosen, g^X mod p can be computed very quickly; therefore, the performance cost is minimized. Additionally, using a fresh key for each handshake provides Perfect Forward Secrecy. Implementations SHOULD generate a new X for each handshake when using DHE cipher suites. Because TLS allows the server to provide arbitrary DH groups, the client should verify that the DH group is of suitable size as defined by local policy. The client SHOULD also verify that the DH public exponent appears to be of adequate size. [KEYSIZ] provides a useful guide to the strength of various group sizes. The server MAY choose to assist the client by providing a known group, such as those defined in [IKEALG] or [MODP]. These can be verified by simple comparison. Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020 For each of the key exchange algorithms, ASICS Digital and/or Google build an executable tamper resistant key module that includes the generated private key and the encrypted data.

	For **RSA and RSA _PSK**, the executable tamper resistant key module includes encrypted data (i.e. the encrypted premaster secret) as it would be impossible for ASICS Digital to send its mobile app to Google using SSL/TLS without encrypting a premaster secret and sending it to Google. The executable tamper resistant key module also includes: 1. Google’s RSA private key corresponding to Google’s RSA public key used by ASICS Digital to encrypt the premaster secret. 2. ASICS Digital’s private key used to code-sign the mobile app. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, the executable tamper resistant key module includes encrypted data (i.e. all communication with Google subsequent to the TLS handshake, including at least the executable compiled code related to its mobile apps and information such as name, category, screenshots and description related to ASICS Digital’s mobile apps). The executable tamper resistant key module also includes: 1. Google’s Diffie-Hellman private key corresponding to Google’s Diffie-Hellman public key used by ASICS Digital to compute the shared master secret. 2. ASICS Digital’s Diffie-Hellman private key used to compute the shared master secret. 3. Google’s RSA private key used to sign the message containing Google’s Diffie-Hellman public key. 4. ASICS Digital’s private key used to code-sign the mobile app. For **DHE_DSS and DH_DSS**, the executable tamper resistant key module includes encrypted data (i.e. all communication with Google subsequent to the TLS handshake, including at least the executable compiled code related to its mobile apps and information such as name, category, screenshots and description related to ASICS Digital’s mobile apps). The executable tamper resistant key module also includes: 1. Google’s Diffie-Hellman private key corresponding to Google’s Diffie-Hellman public key used by ASICS Digital to compute the shared master secret. 2. ASICS Digital’s Diffie-Hellman private key used to compute the shared master secret. 3. Google’s DSA private key used to sign the message containing Google’s Diffie-Hellman public key. 4. ASICS Digital’s private key used to code-sign the mobile app. For **ECDH_ECDSA and ECDHE_ECDSA**, the executable tamper resistant key module includes encrypted data (i.e. all communication with Google subsequent to the TLS handshake, including at least the executable compiled code related to its mobile apps and information such as name, category, screenshots and description related to ASICS Digital’s mobile apps). The executable tamper resistant key module also includes: 1. Google’s Diffie-Hellman private key corresponding to Google’s Diffie-Hellman public key used by ASICS Digital to compute the shared master secret. 2. ASICS Digital’s Diffie-Hellman private key used to compute the shared master secret. 3. Google’s ECDSA private key used to sign the message containing Google’s Diffie-Hellman public key. 4. ASICS Digital’s private key used to code-sign the mobile app.

	ASICS Digital builds an executable tamper resistant key module identified for a selected program resident on a remote system. Specifically, ASICS Digital builds a mobile app, which is an executable tamper resistant key module, as explained in more detail below. This mobile app is identified for a selected program resident on a remote system, namely the Android operating system on a remote mobile device. The mobile app comprises an executable tamper resistant key module that is identified for the Android program and includes the claimed private key described above and the encrypted data encrypted with the claimed public key also described above in building the mobile app so that it can be made available for download from Google servers onto devices compatible with Android operating system for use by customers of ASICS Digital. An asymmetric key pair is used not only to upload the binary code files for the mobile app, but an entire application package, including all of the metadata for the app, such as title, screenshots, and other resources or information such as application type, category, price, *etc*. which are included during the upload process so that the mobile app can be identified by potential users for download[35]. ASICS Digital's mobile app is each an executable tamper resistant key module because it is designed to work with other software, namely the Android operating system as well as other applications or programs installed on a user's mobile device; because the mobile app is resistant to observation and modification, as explained below; and because in building ASICS Digital mobile apps on the Google platform, ASICS Digital's apps include at least the claimed generated private key and the encrypted data including by way of example, the pre-master secret encrypted with the claimed generated public key when the mobile app is securely uploaded onto the Google servers as described above. The tamper resistant key module includes several keys "used for secure communications" per the Court's previous construction including at least the following: For **RSA and RSA _PSK**: 1. Google's RSA private key corresponding to Google's RSA public key used by ASICS Digital to encrypt the premaster secret. 2. ASICS Digital's private key used to code-sign the mobile app. 3. Symmetric key used for uploading the mobile app to Google subsequent to the TLS handshake. 4. Asymmetric keys and symmetric keys used for TLS communications during operation of the app. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**: 1. Google's Diffie-Hellman private key corresponding to Google's Diffie-Hellman public key used by ASICS Digital to compute the shared master secret. 2. ASICS Digital's Diffie-Hellman private key used to compute the shared master secret. 3. Google's RSA private key used to sign the message containing Google's Diffie-Hellman public key. 4. ASICS Digital's private key used to code-sign the mobile app. 5. Shared master secret key used for uploading the mobile app to Google subsequent to the TLS handshake.

[35] *See, e.g.,* https://developer.android.com/studio/publish/index.html, Last accessed on Mar 19, 2020

6. Asymmetric keys and symmetric keys used for TLS communications during operation of the app.

For **DHE_DSS and DH_DSS**:

1. Google’s Diffie-Hellman private key corresponding to Google’s Diffie-Hellman public key used by ASICS Digital to compute the shared master secret.
2. ASICS Digital’s Diffie-Hellman private key used to compute the shared master secret.
3. Google’s DSA private key used to sign the message containing Google’s Diffie-Hellman public key.
4. ASICS Digital’s private key used to code-sign the mobile app.
5. Shared master secret key used for uploading the mobile app to Google subsequent to the TLS handshake.
6. Asymmetric keys and symmetric keys used for TLS communications during operation of the app.

For **ECDH_ECDSA and ECDHE_ECDSA**:

1. Google’s Diffie-Hellman private key corresponding to Google’s Diffie-Hellman public key used by ASICS Digital to compute the shared master secret.
2. ASICS Digital’s Diffie-Hellman private key used to compute the shared master secret.
3. Google’s ECDSA private key used to sign the message containing Google’s Diffie-Hellman public key.
4. ASICS Digital’s private key used to code-sign the mobile app.
5. Shared master secret key used for uploading the mobile app to Google subsequent to the TLS handshake.
6. Asymmetric keys and symmetric keys used for TLS communications during operation of the app.

ASICS Digital’s mobile apps are tamper resistant, resistant to observation and modification, as follows:

1. **Resistant to Observation Because App Source Code Is Compiled Before Upload**

ASICS Digital mobile apps are resistant to observation, at least in part, since ASICS Digital compiles its mobile app source code before submitting the app to Google – and uploads the binary output of the compilation process rather than the source code itself[36].

See, Android Studio Users Guide, “Prepare for Release” stating “To release your application to users you need to create a release-ready package that users can install and run on their Android-powered devices. The release-ready package contains the same components as the debug APK file — compiled source code, resources, manifest file, and so on — and it is built using the same build tools. However, unlike the debug APK file, the

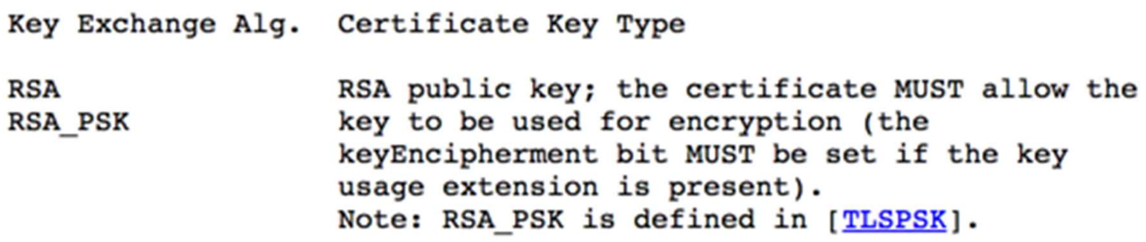
[36] *See, e.g.,* https://developer.android.com/studio/build/index.html, Last accessed on Mar 19, 2020

release-ready APK file is signed with your own certificate and it is optimized with the zipalign tool."
https://developer.android.com/studio/publish/preparing.html, Last accessed on Mar 19, 2020

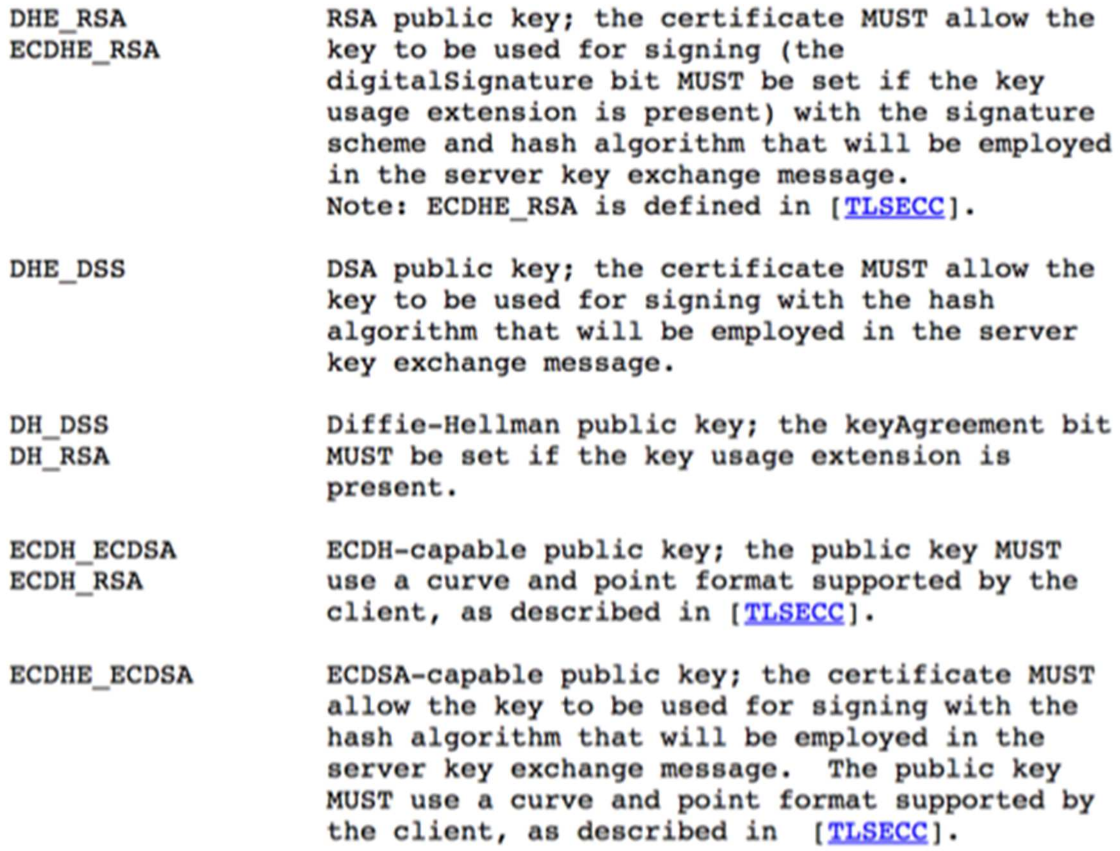
2. Resistant to Observation Because Upload To Google Is Over SSL/TLS

ASICS Digital's mobile apps are made further resistant to observation, at least in part, because the mobile app is securely sent by SSL/TLS to Google as part of the building process. Android Developer Console (https://play.google.com/apps/publish/, Last accessed on Mar 19, 2020) establishes SSL/TLS communications when uploading ASICS Digital's apps, as evidenced by the "https" in the URL. Sending the mobile app code by SSL/TLS is necessary to keep the code from being observed in transit from the code developer to Google.

The secure upload process starts with a TLS handshake procedure which uses asymmetric key encryption and necessitates generation of at least one asymmetric key pair. Specifically, ASICS Digital negotiates with Google the cipher suite and the key exchange algorithm that will be used for the handshake.

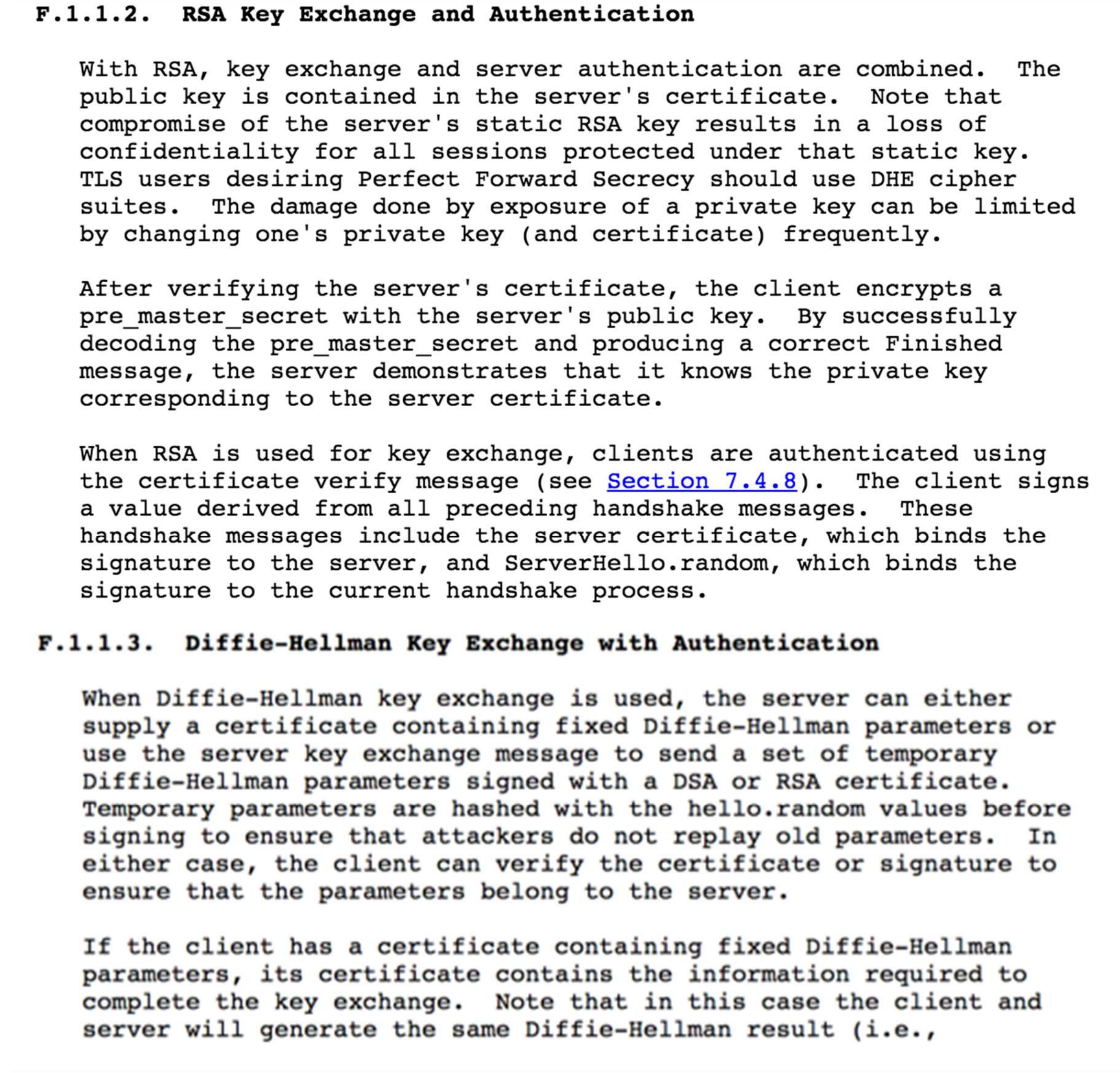


```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].
```



```
DHE_RSA                RSA public key; the certificate MUST allow the
ECDHE_RSA              key to be used for signing (the
                       digitalSignature bit MUST be set if the key
                       usage extension is present) with the signature
                       scheme and hash algorithm that will be employed
                       in the server key exchange message.
                       Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS                DSA public key; the certificate MUST allow the
                       key to be used for signing with the hash
                       algorithm that will be employed in the server
                       key exchange message.

DH_DSS                 Diffie-Hellman public key; the keyAgreement bit
DH_RSA                 MUST be set if the key usage extension is
                       present.

ECDH_ECDSA             ECDH-capable public key; the public key MUST
ECDH_RSA               use a curve and point format supported by the
                       client, as described in [TLSECC].

ECDHE_ECDSA            ECDSA-capable public key; the certificate MUST
                       allow the key to be used for signing with the
                       hash algorithm that will be employed in the
                       server key exchange message.  The public key
                       MUST use a curve and point format supported by
                       the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020



F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

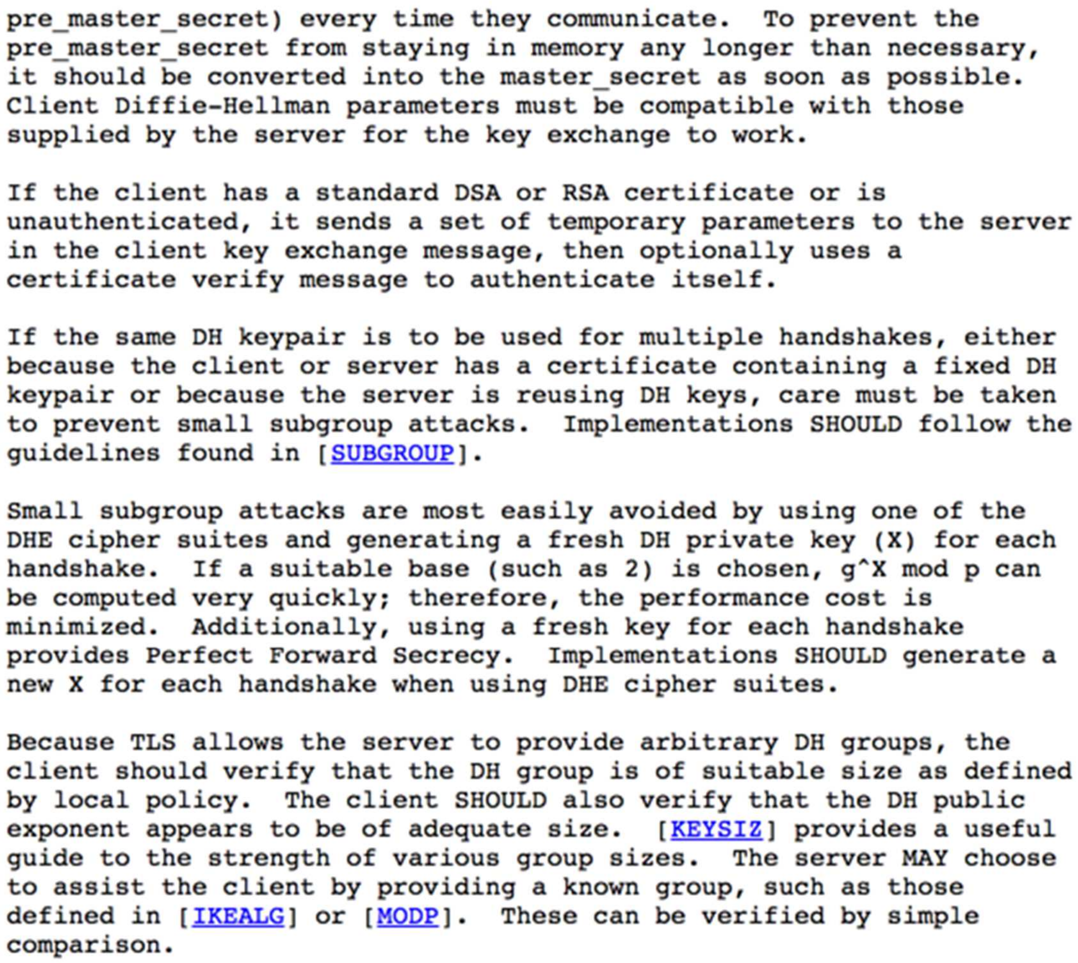
After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.

F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,



pre_master_secret) every time they communicate. To prevent the pre_master_secret from staying in memory any longer than necessary, it should be converted into the master_secret as soon as possible. Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work.

If the client has a standard DSA or RSA certificate or is unauthenticated, it sends a set of temporary parameters to the server in the client key exchange message, then optionally uses a certificate verify message to authenticate itself.

If the same DH keypair is to be used for multiple handshakes, either because the client or server has a certificate containing a fixed DH keypair or because the server is reusing DH keys, care must be taken to prevent small subgroup attacks. Implementations SHOULD follow the guidelines found in [SUBGROUP].

Small subgroup attacks are most easily avoided by using one of the DHE cipher suites and generating a fresh DH private key (X) for each handshake. If a suitable base (such as 2) is chosen, g^X mod p can be computed very quickly; therefore, the performance cost is minimized. Additionally, using a fresh key for each handshake provides Perfect Forward Secrecy. Implementations SHOULD generate a new X for each handshake when using DHE cipher suites.

Because TLS allows the server to provide arbitrary DH groups, the client should verify that the DH group is of suitable size as defined by local policy. The client SHOULD also verify that the DH public exponent appears to be of adequate size. [KEYSIZ] provides a useful guide to the strength of various group sizes. The server MAY choose to assist the client by providing a known group, such as those defined in [IKEALG] or [MODP]. These can be verified by simple comparison.

Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020

For each of the above key exchange algorithms, Google and/or ASICS Digital encrypts all communication, including upload of the mobile app, using a master secret, rendering the communication resistant to observation during transit from ASICS Digital to Google.

For **RSA and RSA _PSK**, ASICS Digital encrypts a random premaster secret with Google's RSA public key and sends the encrypted premaster secret to Google. Google decrypts the premaster secret with its matched RSA private key. ASICS Digital and Google both use the premaster secret to compute a master secret which is then used by both ASICS Digital and Google to encrypt all subsequent communications between ASICS Digital and Google.

For the other key exchange algorithms, namely Diffie-Hellman based algorithms such as **DHE_RSA, ECDHE_RSA, DH_RSA, DHE_DSS, DH_DSS, ECDH_RSA**, **ECDH_ECDSA and ECDHE_ECDSA,** ASICS Digital sends its Diffie-Hellman public key[37] to Google while Google sends its Diffie-Hellman public key to ASICS Digital. ASICS Digital then uses Google's Diffie-Hellman public key combined with ASICS Digital's own Diffie-Hellman private key to compute a premaster secret. Google in turn uses ASICS Digital's Diffie-Hellman public key combined with Google's own Diffie-Hellman private key to compute the same premaster secret. ASICS Digital and Google both use the premaster secret to compute a master secret which is then used by both ASICS Digital and Google to encrypt all subsequent communications between ASICS Digital and Google.

3. Resistant to Modification Because App Binary Is Code Signed

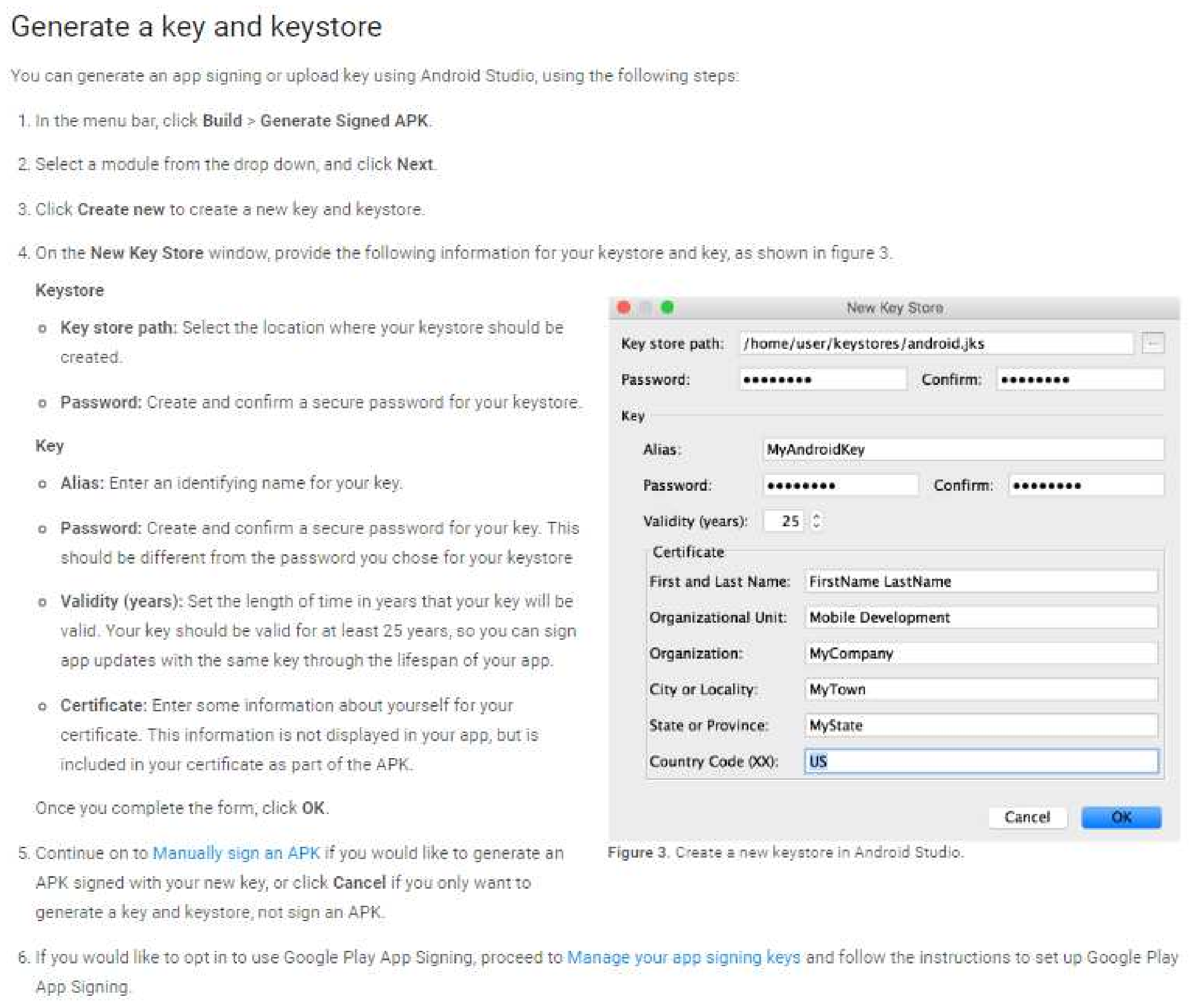
The mobile app code is made resistant to modification, at least in part, because the app binary is code signed. Google dictates that each developer must sign the mobile app submission with his/her asymmetric developer key that certifies that the app has not been modified by a third party impersonator[38].

> *Android requires that the user generates an asymmetric key all apps be digitally signed with a certificate before they can be installed. Android uses this certificate to identify the author of an app*
> Source: https://developer.android.com/studio/publish/app-signing.html, Last accessed on Mar 19, 2020
>
> *A public-key certificate, also known as a digital certificate or an identity certificate, contains the public key of a public/private key pair, as well as some other metadata identifying the owner of the key (for example, name and location). The owner of the certificate holds the corresponding private key.*
> *When you sign an APK, the signing tool attaches the public-key certificate to the APK. The public-key certificate serves as as a "fingerprint" that uniquely associates the APK to you and your corresponding private key. This helps Android ensure that any future updates to your APK are authentic and come from the original author.*
> *A keystore is a binary file that contains one or more private keys. When you sign an APK for release using Android Studio, you can choose to generate a new keystore and private key or use a keystore and private key you already have.*

[37] For Diffie-Hellman (DH) based algorithms such as DH_RSA, DHE_RSA, ECDH_RSA, ECDHE_RSA, DH_DSS, DHE_DSS, ECDH_ECDSA and ECDHE_ECDSA, Google calculates a hash of the message containing their Diffie-Hellman public key and encrypts the hash with their RSA/DSA/ECDSA private key (i.e. signing the message). Google then sends that RSA/DSA/ECDSA public key to ASICS Digital in a digital certificate so that ASICS Digital can authenticate the Google server by decrypting the hash using Google's public key and matching the decryption result to a hash of the received message as calculated by ASICS Digital itself. If the two values match, ASICS Digital knows that the message originated from Google and not from a malicious third party.

[38] *See, e.g.,* https://developer.android.com/tools/publishing/app-signing.html, Last accessed on Mar 19, 2020

Source: https://developer.android.com/studio/publish/app-signing.html , Last accessed on Mar 19, 2020



Generate a key and keystore

You can generate an app signing or upload key using Android Studio, using the following steps:

1. In the menu bar, click **Build** > **Generate Signed APK**.
2. Select a module from the drop down, and click **Next**.
3. Click **Create new** to create a new key and keystore.
4. On the **New Key Store** window, provide the following information for your keystore and key, as shown in figure 3.

   Keystore

   - **Key store path:** Select the location where your keystore should be created.
   - **Password:** Create and confirm a secure password for your keystore.

   Key

   - **Alias:** Enter an identifying name for your key.
   - **Password:** Create and confirm a secure password for your key. This should be different from the password you chose for your keystore
   - **Validity (years):** Set the length of time in years that your key will be valid. Your key should be valid for at least 25 years, so you can sign app updates with the same key through the lifespan of your app.
   - **Certificate:** Enter some information about yourself for your certificate. This information is not displayed in your app, but is included in your certificate as part of the APK.

   Once you complete the form, click **OK**.

5. Continue on to Manually sign an APK if you would like to generate an APK signed with your new key, or click **Cancel** if you only want to generate a key and keystore, not sign an APK.
6. If you would like to opt in to use Google Play App Signing, proceed to Manage your app signing keys and follow the instructions to set up Google Play App Signing.

New Key Store

Key store path: /home/user/keystores/android.jks

Password: •••••••• Confirm: ••••••••

Key

Alias: MyAndroidKey

Password: •••••••• Confirm: ••••••••

Validity (years): 25

Certificate

First and Last Name: FirstName LastName

Organizational Unit: Mobile Development

Organization: MyCompany

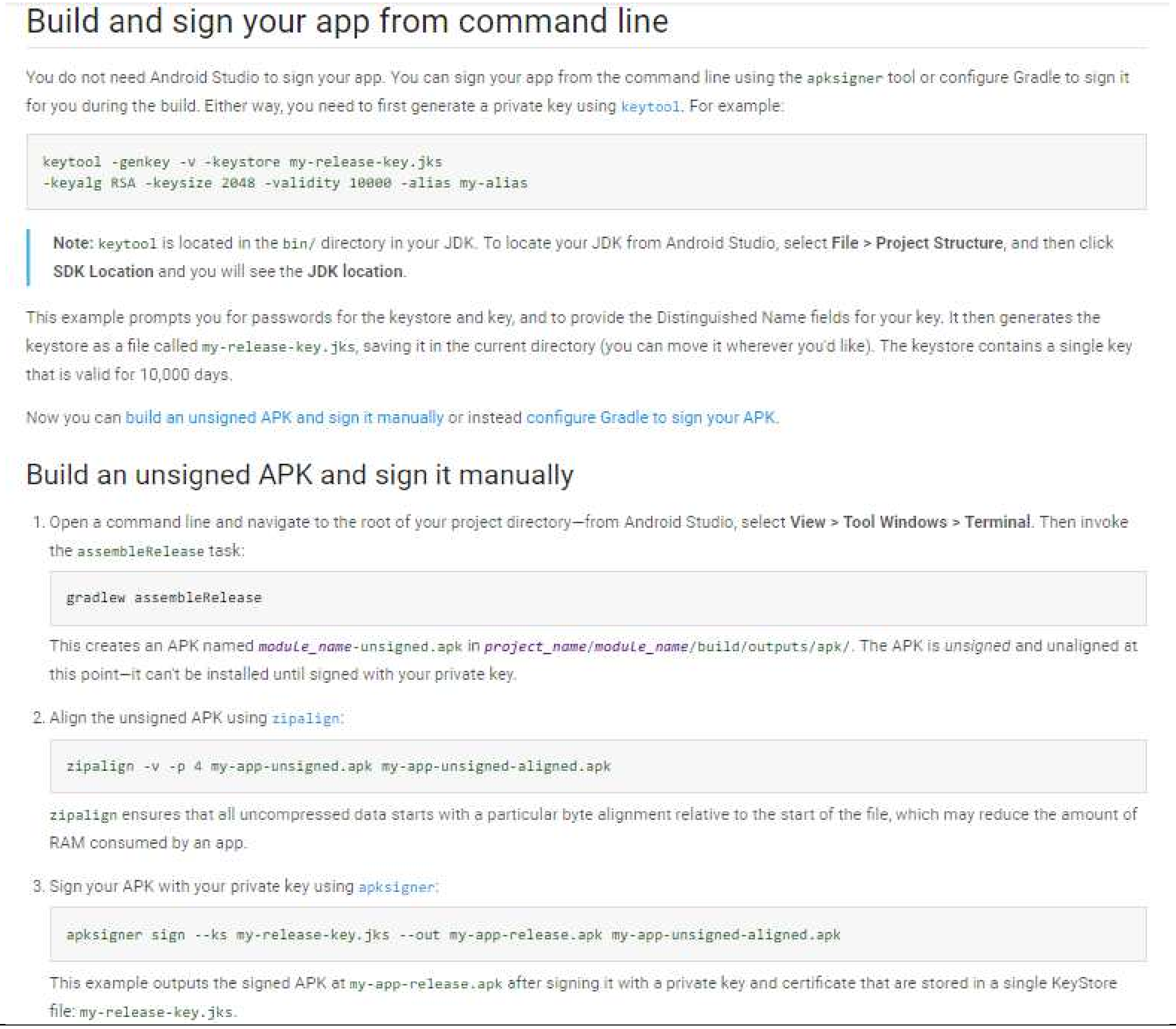
City or Locality: MyTown

State or Province: MyState

Country Code (XX): US

Figure 3. Create a new keystore in Android Studio.

Source: http://web.archive.org/web/20171107004101/https://developer.android.com/studio/publish/app-signing.html#sign-apk, Last accessed on Mar 19, 2020



Build and sign your app from command line

You do not need Android Studio to sign your app. You can sign your app from the command line using the `apksigner` tool or configure Gradle to sign it for you during the build. Either way, you need to first generate a private key using `keytool`. For example:

```
keytool -genkey -v -keystore my-release-key.jks
-keyalg RSA -keysize 2048 -validity 10000 -alias my-alias
```

> **Note:** `keytool` is located in the `bin/` directory in your JDK. To locate your JDK from Android Studio, select **File > Project Structure**, and then click **SDK Location** and you will see the **JDK location**.

This example prompts you for passwords for the keystore and key, and to provide the Distinguished Name fields for your key. It then generates the keystore as a file called `my-release-key.jks`, saving it in the current directory (you can move it wherever you'd like). The keystore contains a single key that is valid for 10,000 days.

Now you can build an unsigned APK and sign it manually or instead configure Gradle to sign your APK.

Build an unsigned APK and sign it manually

1. Open a command line and navigate to the root of your project directory—from Android Studio, select **View > Tool Windows > Terminal**. Then invoke the `assembleRelease` task:

   ```
   gradlew assembleRelease
   ```

   This creates an APK named *`module_name`*`-unsigned.apk` in *`project_name/module_name`*`/build/outputs/apk/`. The APK is *unsigned* and unaligned at this point—it can't be installed until signed with your private key.

2. Align the unsigned APK using `zipalign`:

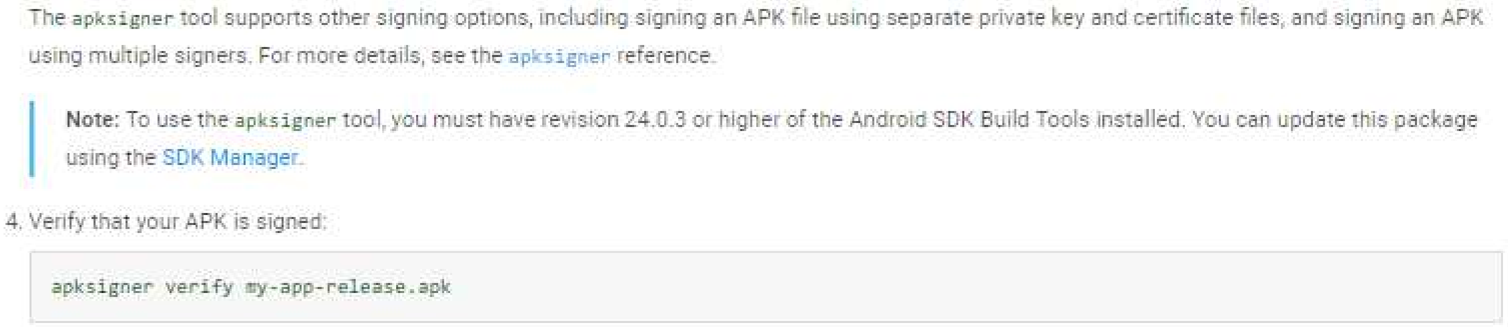
   ```
   zipalign -v -p 4 my-app-unsigned.apk my-app-unsigned-aligned.apk
   ```

   `zipalign` ensures that all uncompressed data starts with a particular byte alignment relative to the start of the file, which may reduce the amount of RAM consumed by an app.

3. Sign your APK with your private key using `apksigner`:

   ```
   apksigner sign --ks my-release-key.jks --out my-app-release.apk my-app-unsigned-aligned.apk
   ```

   This example outputs the signed APK at `my-app-release.apk` after signing it with a private key and certificate that are stored in a single KeyStore file: `my-release-key.jks`.



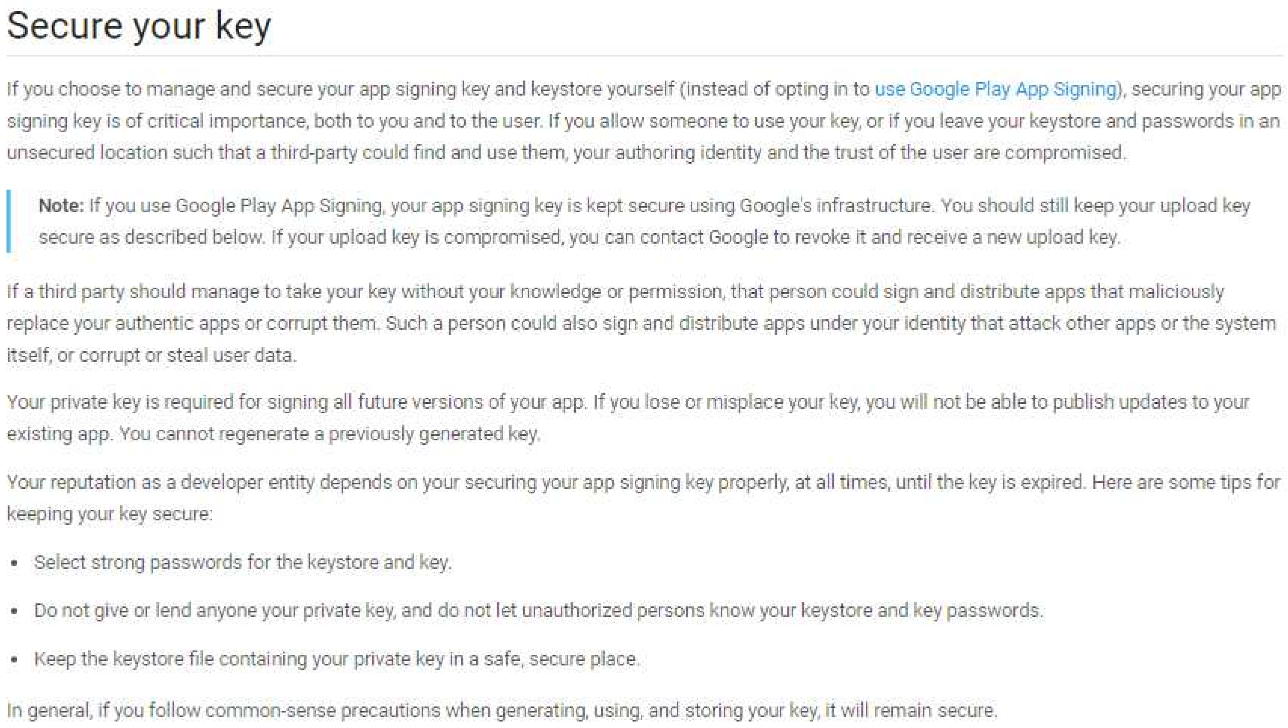
The `apksigner` tool supports other signing options, including signing an APK file using separate private key and certificate files, and signing an APK using multiple signers. For more details, see the `apksigner` reference.

> **Note:** To use the `apksigner` tool, you must have revision 24.0.3 or higher of the Android SDK Build Tools installed. You can update this package using the SDK Manager.

4. Verify that your APK is signed:

```
apksigner verify my-app-release.apk
```

Source: http://web.archive.org/web/20171107004101/https://developer.android.com/studio/publish/app-signing.html#sign-apk, Last accessed on Mar 19, 2020



Secure your key

If you choose to manage and secure your app signing key and keystore yourself (instead of opting in to use Google Play App Signing), securing your app signing key is of critical importance, both to you and to the user. If you allow someone to use your key, or if you leave your keystore and passwords in an unsecured location such that a third-party could find and use them, your authoring identity and the trust of the user are compromised.

> **Note:** If you use Google Play App Signing, your app signing key is kept secure using Google's infrastructure. You should still keep your upload key secure as described below. If your upload key is compromised, you can contact Google to revoke it and receive a new upload key.

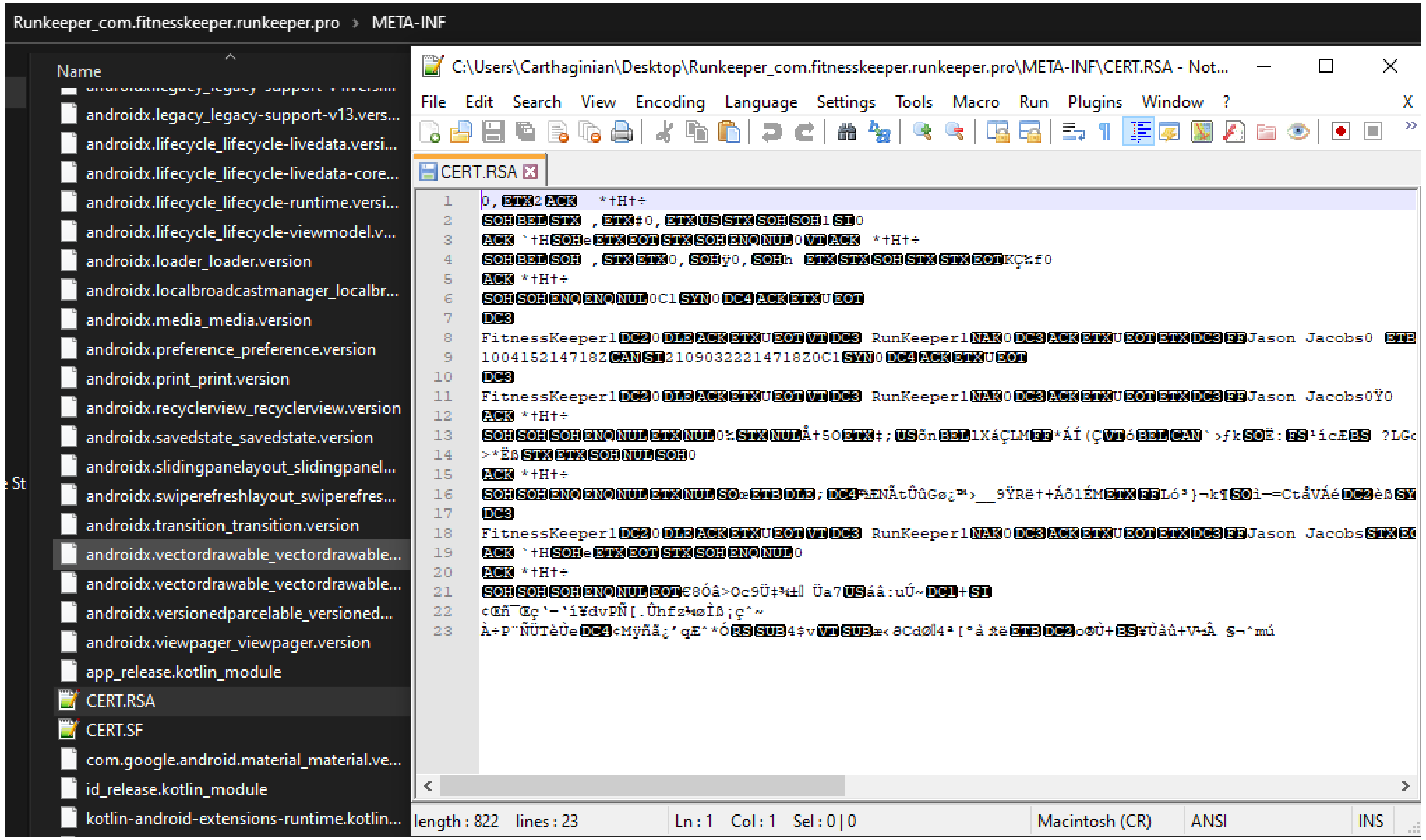
If a third party should manage to take your key without your knowledge or permission, that person could sign and distribute apps that maliciously replace your authentic apps or corrupt them. Such a person could also sign and distribute apps under your identity that attack other apps or the system itself, or corrupt or steal user data.

Your private key is required for signing all future versions of your app. If you lose or misplace your key, you will not be able to publish updates to your existing app. You cannot regenerate a previously generated key.

Your reputation as a developer entity depends on your securing your app signing key properly, at all times, until the key is expired. Here are some tips for keeping your key secure:

- Select strong passwords for the keystore and key.
- Do not give or lend anyone your private key, and do not let unauthorized persons know your keystore and key passwords.
- Keep the keystore file containing your private key in a safe, secure place.

In general, if you follow common-sense precautions when generating, using, and storing your key, it will remain secure.

	Source: http://web.archive.org/web/20171107004101/https://developer.android.com/studio/publish/app-signing.html#sign-apk , Last accessed on Mar 19, 2020 ASICS Digital complies with Google's instructions on code signing as shown by ASICS Digital's mobile app contents. ASICS Digital's mobile apps contain files such as the files CERT.SF and CERT.RSA in ASICS Digital's Android apps which are generated during the code signing process as per instructions from Google. 

Runkeeper_com.fitnesskeeper.runkeeper.pro › META-INF

Name

- androidx.legacy_legacy-support-v...
- androidx.legacy_legacy-support-v13.vers...
- androidx.lifecycle_lifecycle-livedata.versi...
- androidx.lifecycle_lifecycle-livedata-core...
- androidx.lifecycle_lifecycle-runtime.versi...
- androidx.lifecycle_lifecycle-viewmodel.v...
- androidx.loader_loader.version
- androidx.localbroadcastmanager_localbr...
- androidx.media_media.version
- androidx.preference_preference.version
- androidx.print_print.version
- androidx.recyclerview_recyclerview.version
- androidx.savedstate_savedstate.version
- androidx.slidingpanelayout_slidingpanel...
- androidx.swiperefreshlayout_swiperefres...
- androidx.transition_transition.version
- androidx.vectordrawable_vectordrawable...
- androidx.vectordrawable_vectordrawable...
- androidx.versionedparcelable_versioned...
- androidx.viewpager_viewpager.version
- app_release.kotlin_module
- CERT.RSA
- CERT.SF
- com.google.android.material_material.ve...
- id_release.kotlin_module
- kotlin-android-extensions-runtime.kotlin...

St

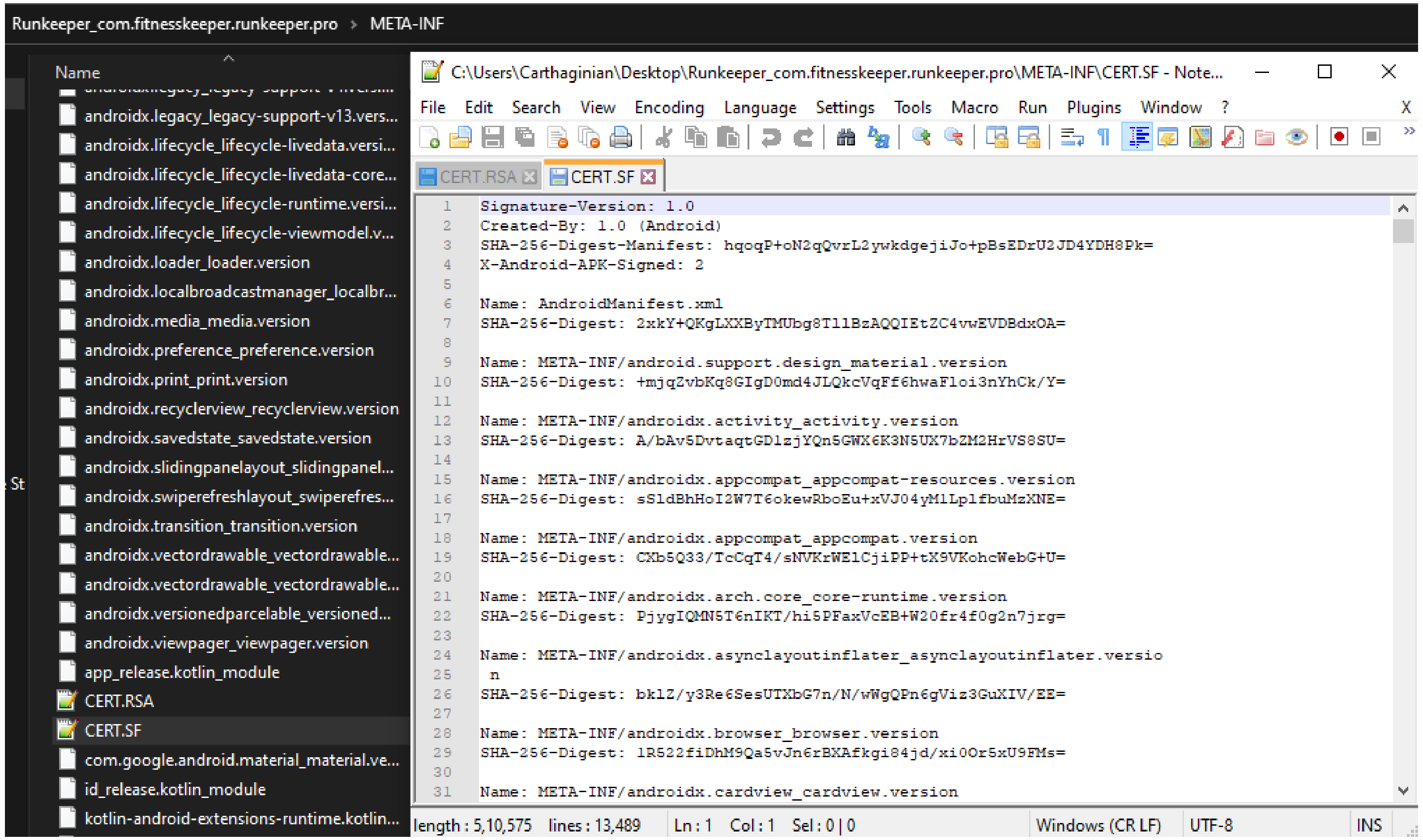
C:\Users\Carthaginian\Desktop\Runkeeper_com.fitnesskeeper.runkeeper.pro\META-INF\CERT.RSA - Not...

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
X
CERT.RSA

```
0,ETX2ACK   *†H†÷
SOHBELSTX ,ETX‡0,ETXUSSTXSOHSOH1SI0
ACK `†HSOHeETXEOTSTXSOHENQNUL0VTACK  *†H†÷
SOHBELSOH ,STXETX0,SOHÿ0,SOHh ETXSTXSOHSTXSTXEOTKÇ‰f0
ACK *†H†÷
SOHSOHENQENQNUL0C1SYN0DC4ACKETXUEOT
DC3
FitnessKeeper1DC20DLEACKETXUEOTVTDC3 RunKeeper1NAK0DC3ACKETXUEOTETXDC3FFJason Jacobs0 ETB
100415214718ZCANSI210903222214718Z0C1SYN0DC4ACKETXUEOT
DC3
FitnessKeeper1DC20DLEACKETXUEOTVTDC3 RunKeeper1NAK0DC3ACKETXUEOTETXDC3FFJason Jacobs0Ÿ0
ACK *†H†÷
SOHSOHSOHENQNULETXNUL0‰STXNULÅ+5OETX‡;USõnBEL1XáÇLMFF*ÁÍ(ÇVTóBELCAN`›fkSOË:FS¹icÆBS ?LGc
>*ËßSTXETXSOHNULSOH0
ACK *†H†÷
SOHSOHENQENQNULETXNULSOæETBDLE;DC4‰ENÃtÛûGø¿™›__9ŸRë†+Áõ1ÉMETXFFLó³}¬k¶SOì—=CtåVÁéDC2èßSY
DC3
FitnessKeeper1DC20DLEACKETXUEOTVTDC3 RunKeeper1NAK0DC3ACKETXUEOTETXDC3FFJason JacobsSTXE
ACK `†HSOHeETXEOTSTXSOHENQNUL0
ACK *†H†÷
SOHSOHSOHENQNULEOT€8Óâ>Oc9Ü‡¾±| Üa7USáâ:uŰ~DC1+SI
¢Œñ¯Œç'–'í¥dvPÑ[.Ûhfz¼øÌß¡ç^~
À÷P¨ÑÜTèÙeDC4¢Mÿñã¿'qE^*ÓRSSUB4$vVTSUBæ‹ðCdØ|4ª[°à ‰ëETBDC2o®Ù+BS¥Ùàû+V¼Â §¬^mú
```

length : 822 lines : 23 Ln : 1 Col : 1 Sel : 0 | 0 Macintosh (CR) ANSI INS

Source: Contents of Runkeeper - GPS Track Run Walk (https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro, Last accessed on Mar 19, 2020) as an example of ASICS Digital app


Runkeeper_com.fitnesskeeper.runkeeper.pro › META-INF
Name
C:\Users\Carthaginian\Desktop\Runkeeper_com.fitnesskeeper.runkeeper.pro\META-INF\CERT.SF - Note...
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
CERT.RSA
CERT.SF
St

androidx.legacy_legacy-support-v13.vers...
androidx.lifecycle_lifecycle-livedata.versi...
androidx.lifecycle_lifecycle-livedata-core...
androidx.lifecycle_lifecycle-runtime.versi...
androidx.lifecycle_lifecycle-viewmodel.v...
androidx.loader_loader.version
androidx.localbroadcastmanager_localbr...
androidx.media_media.version
androidx.preference_preference.version
androidx.print_print.version
androidx.recyclerview_recyclerview.version
androidx.savedstate_savedstate.version
androidx.slidingpanelayout_slidingpanel...
androidx.swiperefreshlayout_swiperefres...
androidx.transition_transition.version
androidx.vectordrawable_vectordrawable...
androidx.vectordrawable_vectordrawable...
androidx.versionedparcelable_versioned...
androidx.viewpager_viewpager.version
app_release.kotlin_module
CERT.RSA
CERT.SF
com.google.android.material_material.ve...
id_release.kotlin_module
kotlin-android-extensions-runtime.kotlin...

```
Signature-Version: 1.0
Created-By: 1.0 (Android)
SHA-256-Digest-Manifest: hqoqP+oN2qQvrL2ywkdgejiJo+pBsEDrU2JD4YDH8Pk=
X-Android-APK-Signed: 2

Name: AndroidManifest.xml
SHA-256-Digest: 2xkY+QKgLXXByTMUbg8T1lBzAQQIEtZC4vwEVDBdxOA=

Name: META-INF/android.support.design_material.version
SHA-256-Digest: +mjqZvbKq8GIgD0md4JLQkcVqFf6hwaFloi3nYhCk/Y=

Name: META-INF/androidx.activity_activity.version
SHA-256-Digest: A/bAv5DvtaqtGDlzjYQn5GWX6K3N5UX7bZM2HrVS8SU=

Name: META-INF/androidx.appcompat_appcompat-resources.version
SHA-256-Digest: sSldBhHoI2W7T6okewRboEu+xVJ04yMlLplfbuMzXNE=

Name: META-INF/androidx.appcompat_appcompat.version
SHA-256-Digest: CXb5Q33/TcCqT4/sNVKrWElCjiPP+tX9VKohcWebG+U=

Name: META-INF/androidx.arch.core_core-runtime.version
SHA-256-Digest: PjygIQMN5T6nIKT/hi5PFaxVcEB+W20fr4f0g2n7jrg=

Name: META-INF/androidx.asynclayoutinflater_asynclayoutinflater.versio
 n
SHA-256-Digest: bklZ/y3Re6SesUTXbG7n/N/wWgQPn6gViz3GuXIV/EE=

Name: META-INF/androidx.browser_browser.version
SHA-256-Digest: lR522fiDhM9Qa5vJn6rBXAfkgi84jd/xi0Or5xU9FMs=

Name: META-INF/androidx.cardview_cardview.version
```

length : 5,10,575 lines : 13,489 Ln : 1 Col : 1 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

Source: Contents of Runkeeper - GPS Track Run Walk (https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro, Last accessed on Mar 19, 2020) as an example of ASICS Digital app

	Asymmetrical key cryptography and hashing algorithms are used to create the unique digital signature for Android mobile apps. The digital signature is used to sign the resources in an application package, including the compiled code. The private key of an asymmetric key pair that is generated for the digital code signing is used to code sign the app. This private key is included in the mobile app although the private key is not the claimed private key of the claimed generated asymmetric key pair because it does not match the claimed public key used to encrypt data, based on the court's construction. Hashes are created for every resource in the application package with the help of a hash algorithm. The signature manifest also has its own hash to prevent unauthorized changes. The hashes are encrypted with a private key. After the encryption is complete, the digital signature for the app is created. By signing the app binary with a digital signature, ASICS Digital's mobile apps are tamper resistant enabling Google and the Android mobile devices to verify that the application is being distributed by trusted source (*i.e.* ASICS Digital) and that the application has not been modified by a third party, which can be verified by the corresponding public key generated as part of the pair. Thus the app binary is made resistant to modification by digital signing. Accordingly ASICS Digital's Android mobile apps establish SSL/TLS communications with ASICS Digital's servers, which involve a SSL/TLS handshake procedure involving asymmetric key encryption. SSL/TLS ensures secure communication and renders the mobile app data further resistant to observation.
sending the executable tamper resistant key module to the remote system.	Upon information and belief, the method step of sending the executable tamper resistant key module to the remote system is performed by Google and/or its agents – whose acts are attributable to ASICS Digital (i) because ASICS Digital works together with Google in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Google distributes and markets ASICS Digital's mobile apps under the direction and control of ASICS Digital, or acts as agent, or on behalf of ASICS Digital, in the building, marketing and distribution of ASICS Digital's mobile apps. Alternatively, to the extent any portion of this method step is performed by ASICS Digital, such acts are attributable to Google, who conditions participation in and the receipt of a benefit, namely, the distribution of ASICS Digital's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Google in the building and upload of ASICS Digital's mobile apps, and ASICS Digital induces infringement by Google in the building, marketing and distribution of ASICS Digital's mobile apps. ASICS Digital's mobile apps are sent or downloaded from Google servers and are executed on Android remote devices such as mobile phones and tablets. When a user accesses Google Play Store – and requests to download ASICS Digital app, Google sends the executable tamper resistant key module from the Google servers to the remote device(s).

Further, the step of “sending” ASICS Digital mobile apps to the remote system occurs via TLS/SSL communications. Thus, the sending of ASICS Digital mobile apps, to the extent required by the claims, also includes a private key and data encrypted by a public key, as explained in detail above.

In particular, ASICS Digital mobile apps sent to users’ remote devices are tamper resistant, resistant to observation and modification as follows:

1. **Resistant to Observation Because App is Downloaded in Compiled Form**

ASICS Digital mobile apps are resistant to observation, at least in part, since ASICS Digital compiles its mobile app source code before submitting the app to Google – and uploads the binary output of the compilation process rather than the source code itself – and hence a user can only download the compiled source code from Google rather than the source code itself[39].

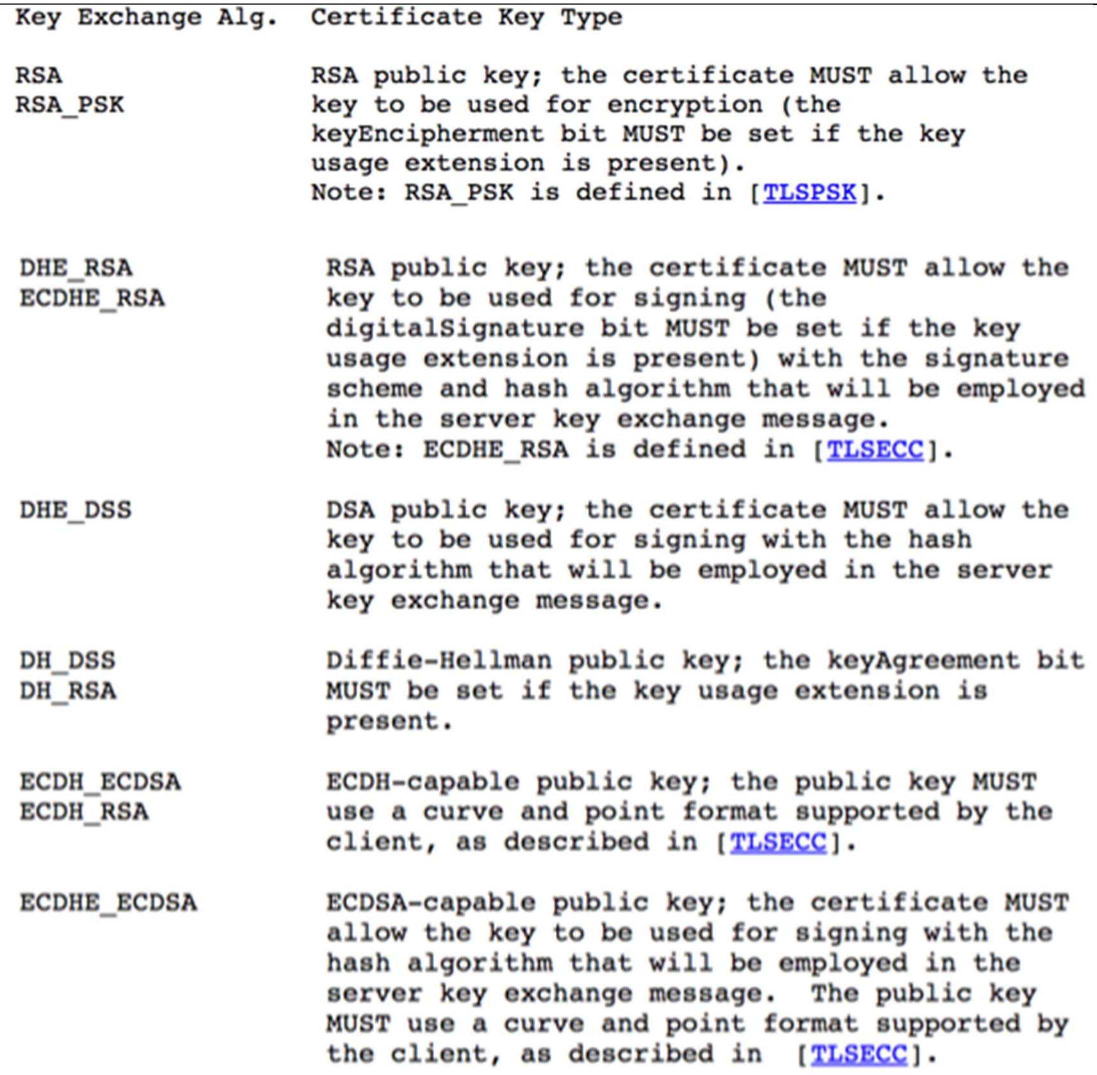
See, Android Studio Users Guide, “Prepare for Release” stating “To release your application to users you need to create a release-ready package that users can install and run on their Android-powered devices. The release-ready package contains the same components as the debug APK file — compiled source code, resources, manifest file, and so on — and it is built using the same build tools. However, unlike the debug APK file, the release-ready APK file is signed with your own certificate and it is optimized with the zipalign tool.” https://developer.android.com/studio/publish/preparing.html, Last accessed on Mar 19, 2020

2. **Resistant to Observation Because Download from Google Is Over SSL/TLS**

ASICS Digital’s mobile apps are made further resistant to observation, at least in part, because the mobile app is securely sent or downloaded by SSL/TLS from Google servers. ASICS Digital app users establish SSL/TLS communications with Play Store (for example using the URL https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro for ASICS Runkeeper, Last accessed on Mar 19, 2020) when downloading ASICS Digital’s apps, as evidenced by the “https” in the URL. Sending the mobile app code by SSL/TLS is necessary to keep the code from being observed in transit from Google to the user’s remote system.

The secure download process starts with a TLS handshake procedure which uses asymmetric key encryption and necessitates generation of at least one asymmetric key pair. Specifically, user’s remote device negotiates with Google the cipher suite and the key exchange algorithm that will be used for the handshake:

[39] *See, e.g.,* https://developer.android.com/studio/build/index.html, Last accessed on Mar 19, 2020

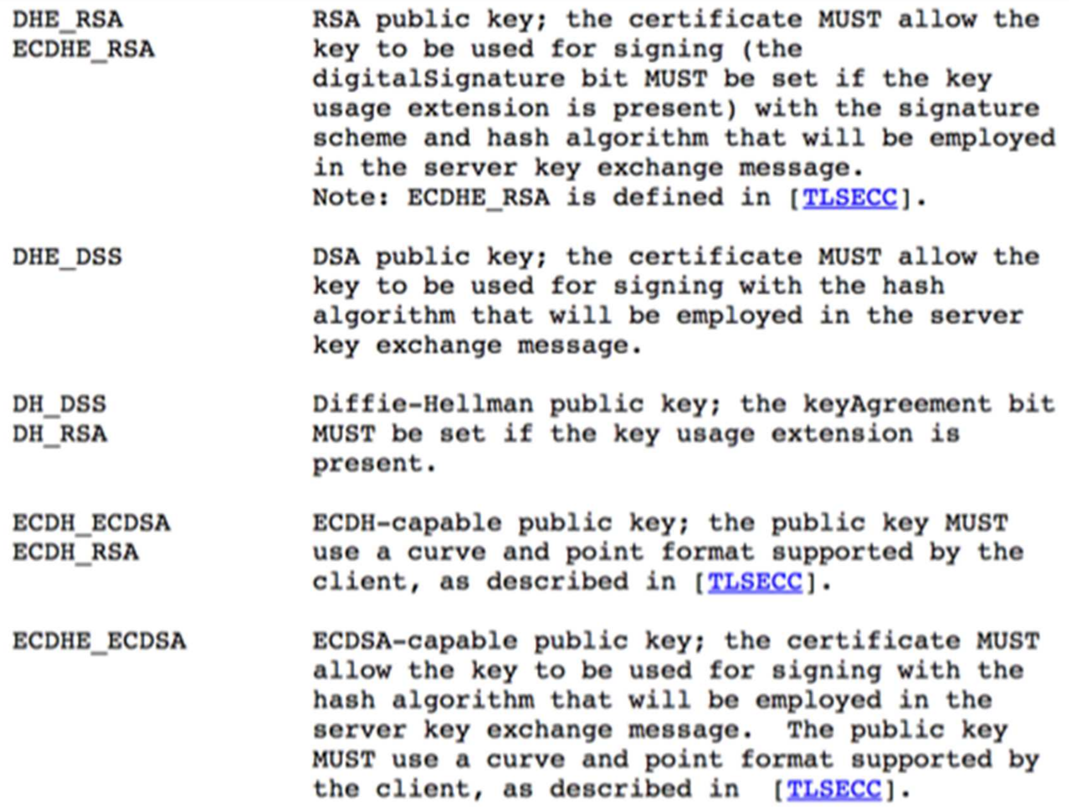


```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].

DHE_RSA            RSA public key; the certificate MUST allow the
ECDHE_RSA          key to be used for signing (the
                   digitalSignature bit MUST be set if the key
                   usage extension is present) with the signature
                   scheme and hash algorithm that will be employed
                   in the server key exchange message.
                   Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS            DSA public key; the certificate MUST allow the
                   key to be used for signing with the hash
                   algorithm that will be employed in the server
                   key exchange message.

DH_DSS             Diffie-Hellman public key; the keyAgreement bit
DH_RSA             MUST be set if the key usage extension is
                   present.

ECDH_ECDSA         ECDH-capable public key; the public key MUST
ECDH_RSA           use a curve and point format supported by the
                   client, as described in [TLSECC].

ECDHE_ECDSA        ECDSA-capable public key; the certificate MUST
                   allow the key to be used for signing with the
                   hash algorithm that will be employed in the
                   server key exchange message.  The public key
                   MUST use a curve and point format supported by
                   the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

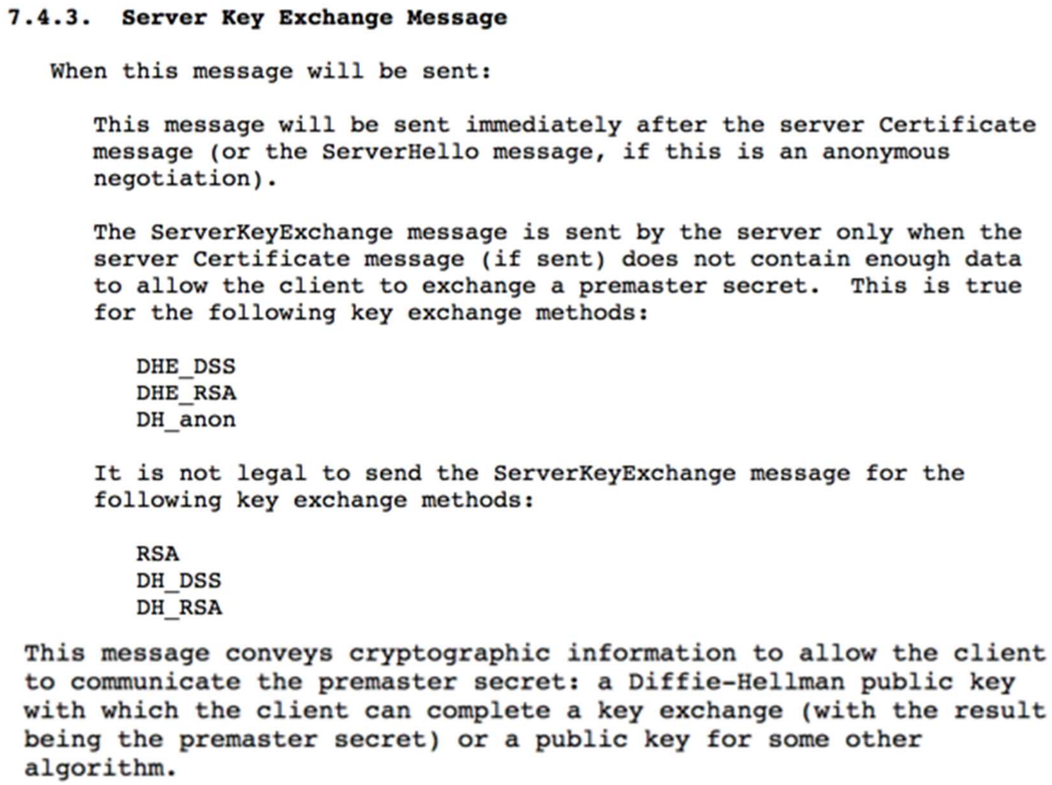
	Each of these algorithms necessitates generating one or more asymmetric key pairs – that are in turn used to compute a shared master secret for encrypting the mobile app download. For **RSA and RSA _PSK**, Google server generates an RSA public-private key pair. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, Google server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[40]. The user's remote device also generates a second Diffie-Hellman public-private key pair. For **DHE_DSS and DH_DSS**, Google server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. For **ECDH_ECDSA and ECDHE_ECDSA**, Google server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair.

[40] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN 1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020



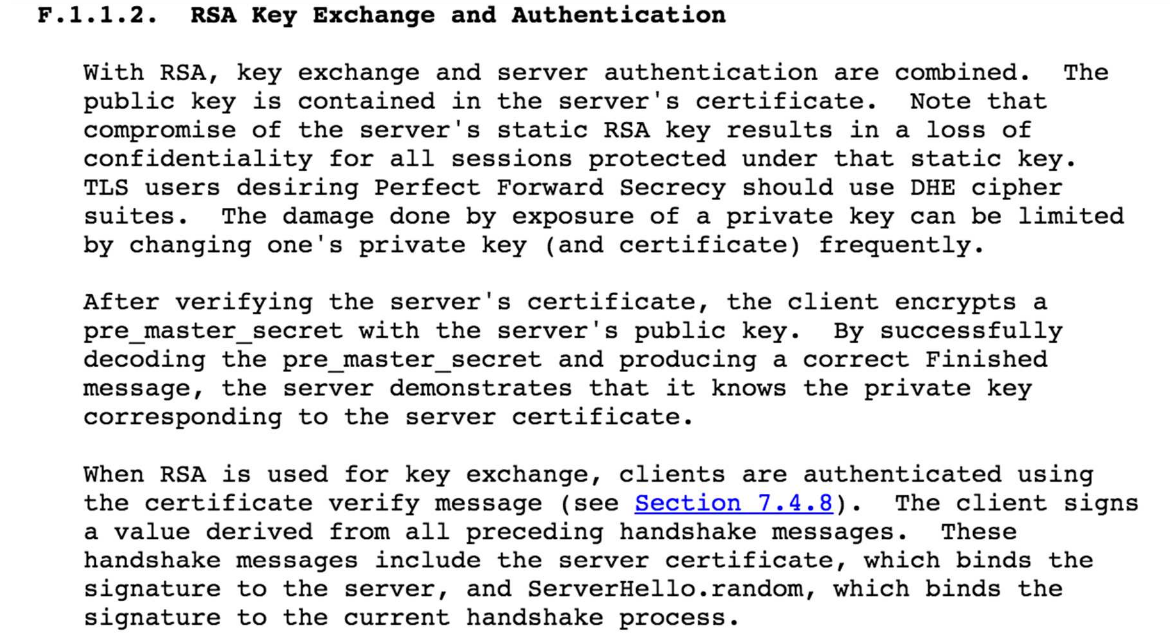
```
DHE_RSA              RSA public key; the certificate MUST allow the
ECDHE_RSA            key to be used for signing (the
                     digitalSignature bit MUST be set if the key
                     usage extension is present) with the signature
                     scheme and hash algorithm that will be employed
                     in the server key exchange message.
                     Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS              DSA public key; the certificate MUST allow the
                     key to be used for signing with the hash
                     algorithm that will be employed in the server
                     key exchange message.

DH_DSS               Diffie-Hellman public key; the keyAgreement bit
DH_RSA               MUST be set if the key usage extension is
                     present.

ECDH_ECDSA           ECDH-capable public key; the public key MUST
ECDH_RSA             use a curve and point format supported by the
                     client, as described in [TLSECC].

ECDHE_ECDSA          ECDSA-capable public key; the certificate MUST
                     allow the key to be used for signing with the
                     hash algorithm that will be employed in the
                     server key exchange message.  The public key
                     MUST use a curve and point format supported by
                     the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020



```
7.4.3.  Server Key Exchange Message

   When this message will be sent:

      This message will be sent immediately after the server Certificate
      message (or the ServerHello message, if this is an anonymous
      negotiation).

      The ServerKeyExchange message is sent by the server only when the
      server Certificate message (if sent) does not contain enough data
      to allow the client to exchange a premaster secret.  This is true
      for the following key exchange methods:

         DHE_DSS
         DHE_RSA
         DH_anon

      It is not legal to send the ServerKeyExchange message for the
      following key exchange methods:

         RSA
         DH_DSS
         DH_RSA

  This message conveys cryptographic information to allow the client
  to communicate the premaster secret: a Diffie-Hellman public key
  with which the client can complete a key exchange (with the result
  being the premaster secret) or a public key for some other
  algorithm.
```

Source: https://tools.ietf.org/html/rfc5246 at 50-51, Last accessed on Mar 19, 2020

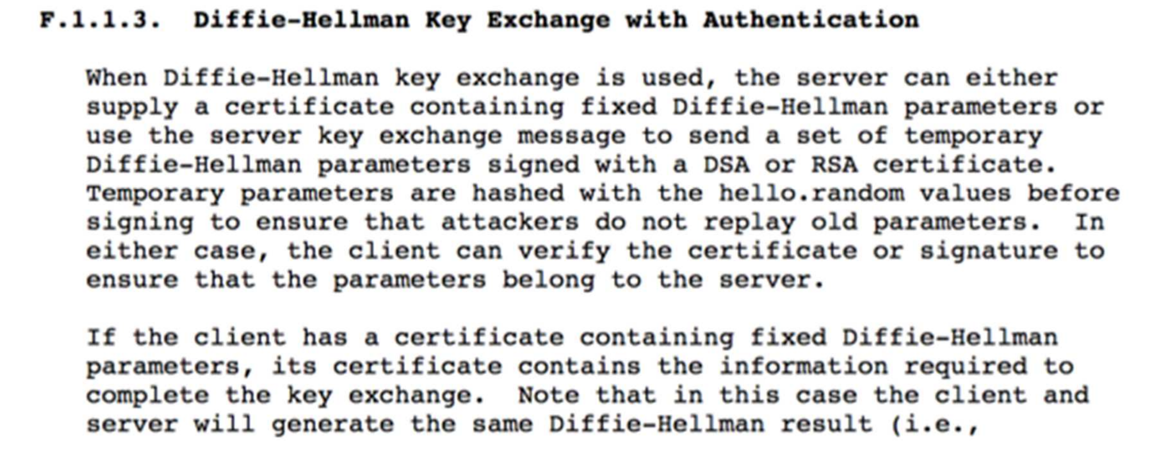


F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

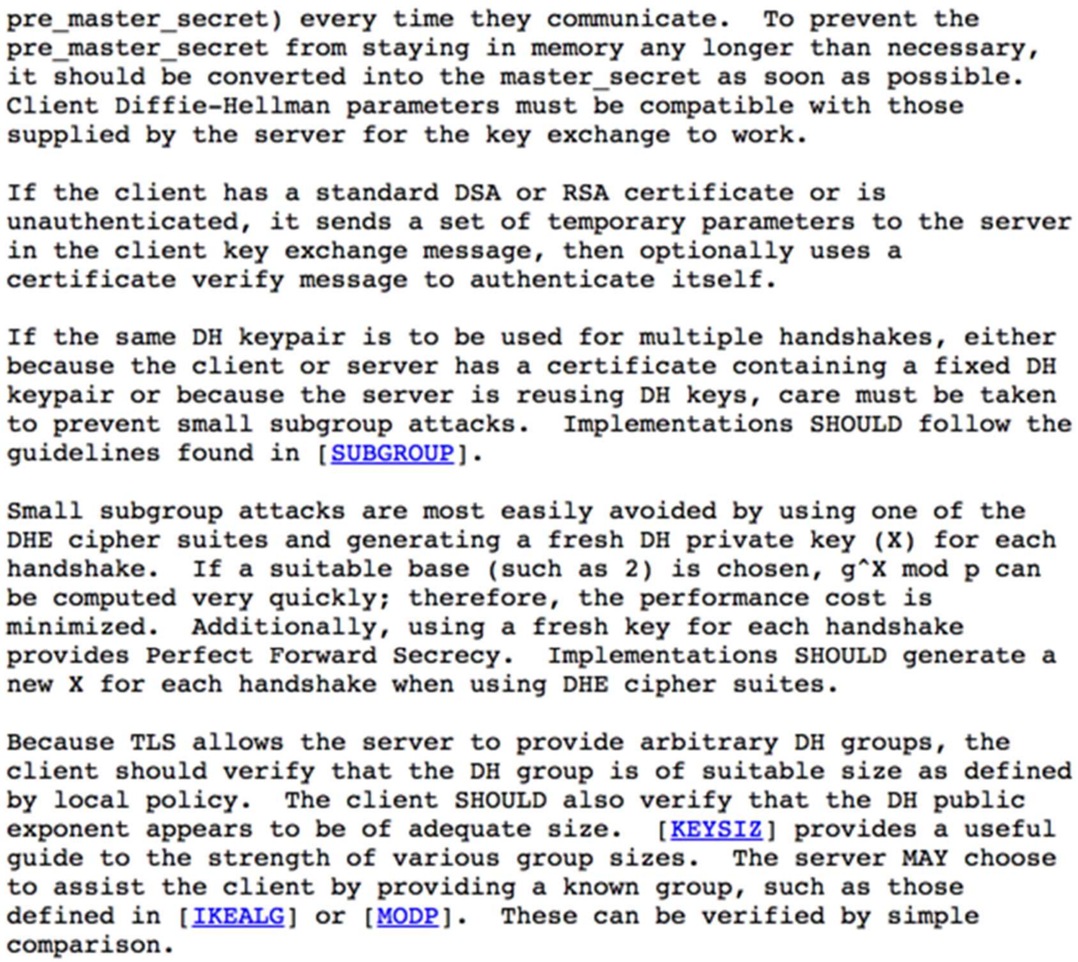
When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.



F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,

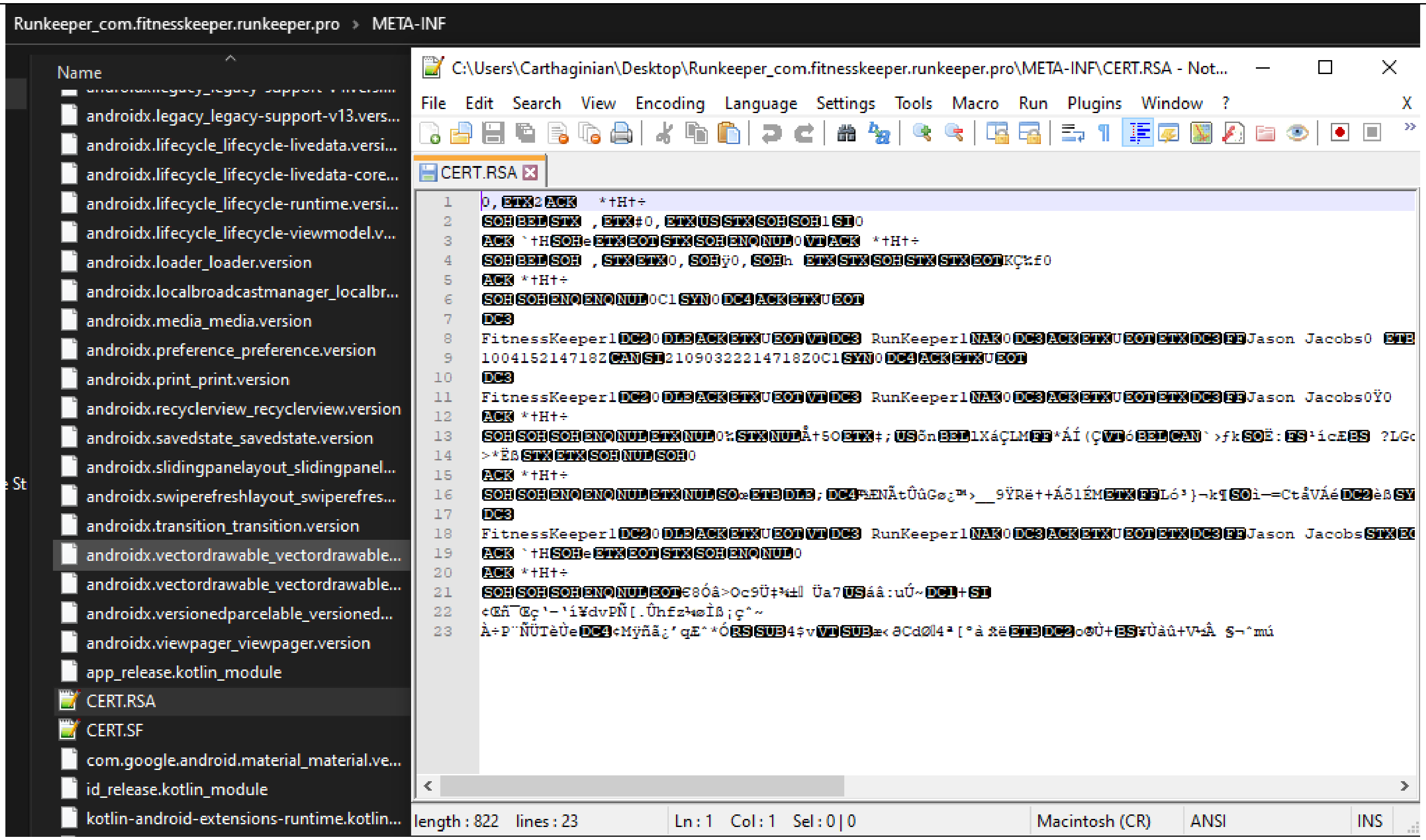
	 pre_master_secret) every time they communicate. To prevent the pre_master_secret from staying in memory any longer than necessary, it should be converted into the master_secret as soon as possible. Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work. If the client has a standard DSA or RSA certificate or is unauthenticated, it sends a set of temporary parameters to the server in the client key exchange message, then optionally uses a certificate verify message to authenticate itself. If the same DH keypair is to be used for multiple handshakes, either because the client or server has a certificate containing a fixed DH keypair or because the server is reusing DH keys, care must be taken to prevent small subgroup attacks. Implementations SHOULD follow the guidelines found in [SUBGROUP]. Small subgroup attacks are most easily avoided by using one of the DHE cipher suites and generating a fresh DH private key (X) for each handshake. If a suitable base (such as 2) is chosen, g^X mod p can be computed very quickly; therefore, the performance cost is minimized. Additionally, using a fresh key for each handshake provides Perfect Forward Secrecy. Implementations SHOULD generate a new X for each handshake when using DHE cipher suites. Because TLS allows the server to provide arbitrary DH groups, the client should verify that the DH group is of suitable size as defined by local policy. The client SHOULD also verify that the DH public exponent appears to be of adequate size. [KEYSIZ] provides a useful guide to the strength of various group sizes. The server MAY choose to assist the client by providing a known group, such as those defined in [IKEALG] or [MODP]. These can be verified by simple comparison. Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020 The generated asymmetric key pairs are then used to compute a shared master secret which is then used to encrypt the mobile app download so that it is resistant to observation during transit.

	For **RSA and RSA _PSK**, the RSA public-private key pair is used to encrypt a random premaster secret which is in turn used by Google server and the user's remote device to compute a master secret. Google uses the master secret to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, Google server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[41]. The user's remote device also generates a second Diffie-Hellman public-private key pair. Google server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Google's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret that Google uses to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. For **DHE_DSS and DH_DSS**, Google server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. Google server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Google's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret that Google uses to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. For **ECDH_ECDSA and ECDHE_ECDSA**, Google server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. Google server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Google's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret that Google uses to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. **3. Resistant to Modification Because Mobile App is Code Signed** The downloaded mobile app code is resistant to modification, at least in part, because the downloaded app binary is code signed. Code-signing allows users' remote systems to verify that the downloaded app binary is authentic and has not been maliciously modified by a third party. Google

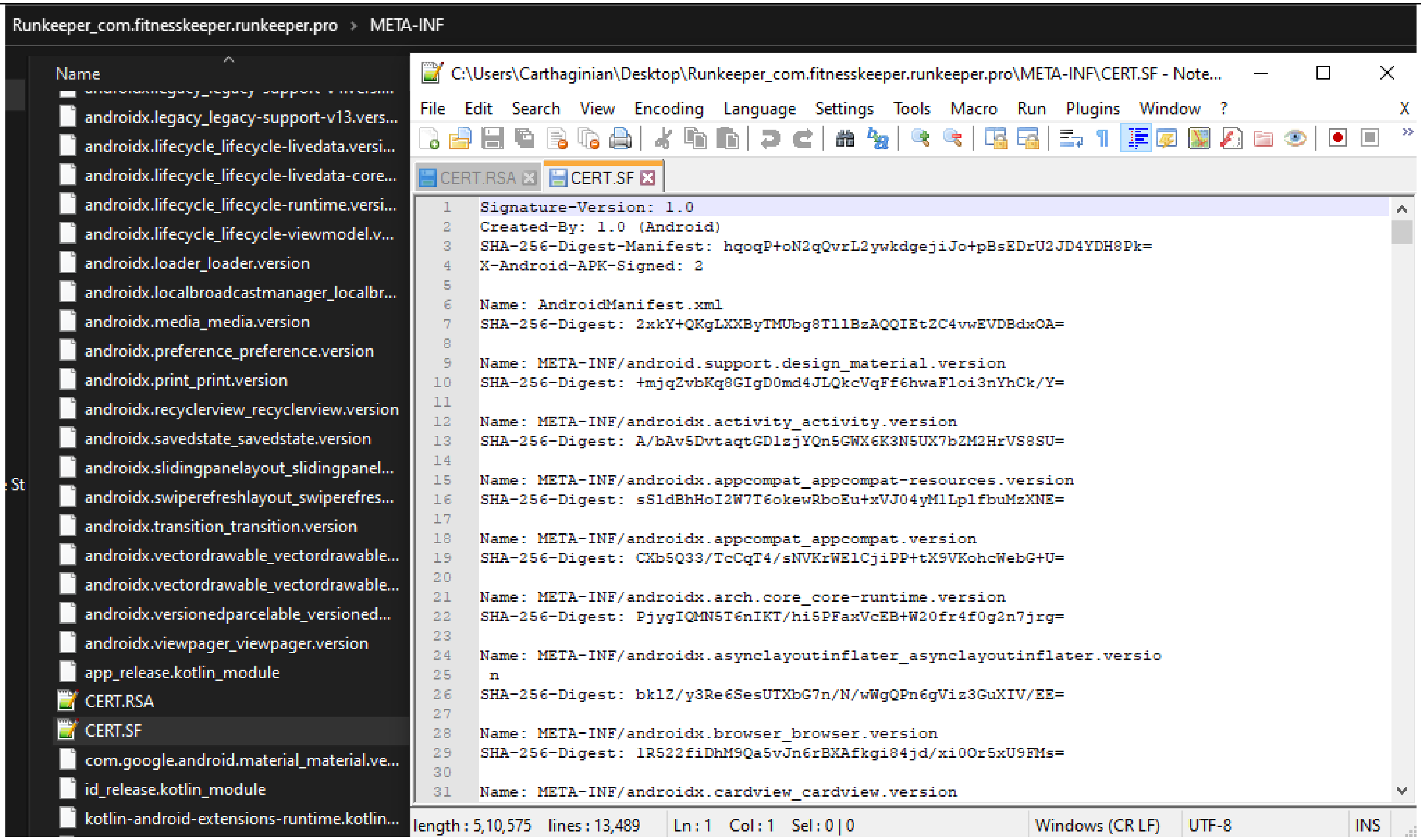
[41] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN 1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020

	dictates that each developer must sign the mobile app submission with his/her asymmetric developer key that certifies that the app has not been modified by a third party impersonator[42]. *Android requires that the user generates an asymmetric key all apps be digitally signed with a certificate before they can be installed. Android uses this certificate to identify the author of an app* Source: https://developer.android.com/studio/publish/app-signing.html, Last accessed on Mar 19, 2020 *A public-key certificate, also known as a digital certificate or an identity certificate, contains the public key of a public/private key pair, as well as some other metadata identifying the owner of the key (for example, name and location). The owner of the certificate holds the corresponding private key.* *When you sign an APK, the signing tool attaches the public-key certificate to the APK. The public-key certificate serves as as a "fingerprint" that uniquely associates the APK to you and your corresponding private key. This helps Android ensure that any future updates to your APK are authentic and come from the original author.* *A keystore is a binary file that contains one or more private keys. When you sign an APK for release using Android Studio, you can choose to generate a new keystore and private key or use a keystore and private key you already have.* Source: https://developer.android.com/studio/publish/app-signing.html , Last accessed on Mar 19, 2020 ASICS Digital complies with Google's instructions on code signing as shown by ASICS Digital's mobile app contents. ASICS Digital's mobile apps contain files such as the files CERT.SF and CERT.RSA in ASICS Digital's Android mobile apps which are generated during the code signing process as per instructions from Google.

[42] *See, e.g.,* https://developer.android.com/tools/publishing/app-signing.html, Last accessed on Mar 19, 2020


Runkeeper_com.fitnesskeeper.runkeeper.pro › META-INF
Name
androidx.legacy_legacy-support-v13.vers...
androidx.lifecycle_lifecycle-livedata.versi...
androidx.lifecycle_lifecycle-livedata-core...
androidx.lifecycle_lifecycle-runtime.versi...
androidx.lifecycle_lifecycle-viewmodel.v...
androidx.loader_loader.version
androidx.localbroadcastmanager_localbr...
androidx.media_media.version
androidx.preference_preference.version
androidx.print_print.version
androidx.recyclerview_recyclerview.version
androidx.savedstate_savedstate.version
androidx.slidingpanelayout_slidingpanel...
androidx.swiperefreshlayout_swiperefres...
androidx.transition_transition.version
androidx.vectordrawable_vectordrawable...
androidx.vectordrawable_vectordrawable...
androidx.versionedparcelable_versioned...
androidx.viewpager_viewpager.version
app_release.kotlin_module
CERT.RSA
CERT.SF
com.google.android.material_material.ve...
id_release.kotlin_module
kotlin-android-extensions-runtime.kotlin...
C:\Users\Carthaginian\Desktop\Runkeeper_com.fitnesskeeper.runkeeper.pro\META-INF\CERT.RSA - Not...
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
CERT.RSA
FitnessKeeper1 RunKeeper1 Jason Jacobs0
100415214718Z 210903222147187Z0
length : 822 lines : 23 Ln : 1 Col : 1 Sel : 0 | 0 Macintosh (CR) ANSI INS

Source: Contents of Runkeeper - GPS Track Run Walk (https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro, Last accessed on Mar 19, 2020) as an example of ASICS Digital app



Runkeeper_com.fitnesskeeper.runkeeper.pro › META-INF
Name
St
C:\Users\Carthaginian\Desktop\Runkeeper_com.fitnesskeeper.runkeeper.pro\META-INF\CERT.SF - Note...
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
CERT.RSA
CERT.SF

androidx.legacy_legacy-support-v13.vers...
androidx.lifecycle_lifecycle-livedata.versi...
androidx.lifecycle_lifecycle-livedata-core...
androidx.lifecycle_lifecycle-runtime.versi...
androidx.lifecycle_lifecycle-viewmodel.v...
androidx.loader_loader.version
androidx.localbroadcastmanager_localbr...
androidx.media_media.version
androidx.preference_preference.version
androidx.print_print.version
androidx.recyclerview_recyclerview.version
androidx.savedstate_savedstate.version
androidx.slidingpanelayout_slidingpanel...
androidx.swiperefreshlayout_swiperefres...
androidx.transition_transition.version
androidx.vectordrawable_vectordrawable...
androidx.vectordrawable_vectordrawable...
androidx.versionedparcelable_versioned...
androidx.viewpager_viewpager.version
app_release.kotlin_module
CERT.RSA
CERT.SF
com.google.android.material_material.ve...
id_release.kotlin_module
kotlin-android-extensions-runtime.kotlin...

```
Signature-Version: 1.0
Created-By: 1.0 (Android)
SHA-256-Digest-Manifest: hqoqP+oN2qQvrL2ywkdgejiJo+pBsEDrU2JD4YDH8Pk=
X-Android-APK-Signed: 2

Name: AndroidManifest.xml
SHA-256-Digest: 2xkY+QKgLXXByTMUbg8T1lBzAQQIEtZC4vwEVDBdxOA=

Name: META-INF/android.support.design_material.version
SHA-256-Digest: +mjqZvbKq8GIgD0md4JLQkcVqFf6hwaFloi3nYhCk/Y=

Name: META-INF/androidx.activity_activity.version
SHA-256-Digest: A/bAv5DvtaqtGDlzjYQn5GWX6K3N5UX7bZM2HrVS8SU=

Name: META-INF/androidx.appcompat_appcompat-resources.version
SHA-256-Digest: sSldBhHoI2W7T6okewRboEu+xVJ04yMlLplfbuMzXNE=

Name: META-INF/androidx.appcompat_appcompat.version
SHA-256-Digest: CXb5Q33/TcCqT4/sNVKrWElCjiPP+tX9VKohcWebG+U=

Name: META-INF/androidx.arch.core_core-runtime.version
SHA-256-Digest: PjygIQMN5T6nIKT/hi5PFaxVcEB+W20fr4f0g2n7jrg=

Name: META-INF/androidx.asynclayoutinflater_asynclayoutinflater.versio
 n
SHA-256-Digest: bklZ/y3Re6SesUTXbG7n/N/wWgQPn6gViz3GuXIV/EE=

Name: META-INF/androidx.browser_browser.version
SHA-256-Digest: 1R522fiDhM9Qa5vJn6rBXAfkgi84jd/xi0Or5xU9FMs=

Name: META-INF/androidx.cardview_cardview.version
```

length : 5,10,575   lines : 13,489   Ln : 1   Col : 1   Sel : 0 | 0   Windows (CR LF)   UTF-8   INS

Source: Contents of Runkeeper - GPS Track Run Walk (https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro, Last accessed on Mar 19, 2020) as an example of ASICS Digital app

By signing the app binary with a digital signature, ASICS Digital's mobile apps are tamper resistant enabling Google and the Android mobile devices to verify that the application is being distributed by trusted source (*i.e.* ASICS Digital) and that the application has not been modified by a third party, which can be verified by the corresponding public key generated as part of the pair. Thus the app binary is made resistant to modification by digital signing.

Accordingly ASICS Digital's Android mobile apps establish SSL/TLS communications with ASICS Digital's servers, which involve a SSL/TLS handshake procedure involving asymmetric key encryption. SSL/TLS ensures secure communication and renders the mobile app data further resistant to observation.

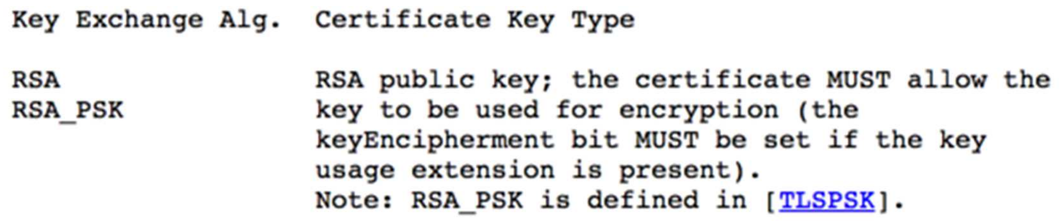
4. Resistant to Observation Because Mobile App is Stored on Remote System in Encrypted Form

The mobile app is made further resistant to observation because when downloaded and installed on a user's Android mobile device, it is stored in an encrypted form. Android implements disk encryption for encrypting the operating system software, apps and all related data on a mobile device – which further renders ASICS Digital app resistant to observation[43].

5. Resistant to Observation Because Mobile App Securely Communicates with ASICS Digital Over SSL/TLS

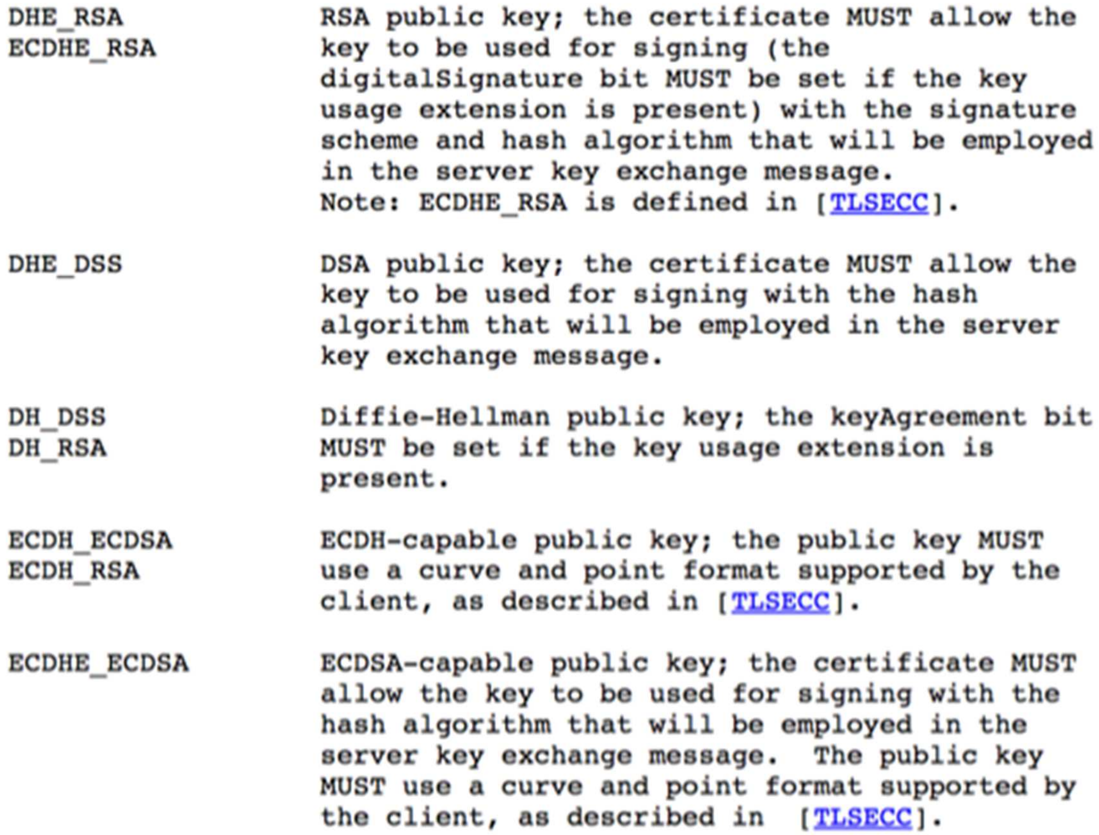
ASICS Digital's mobile apps are made further resistant to observation, at least in part, because the mobile app communicates with ASICS Digital using SSL/TLS during operation. ASICS Digital app users establish SSL/TLS communications with ASICS Digital servers when the app is executed. Such secure communication is necessary to keep source code as well as user identity and activity from being observed in transit from the remote system to ASICS Digital servers and vice versa.

The secure communications process starts with a TLS handshake procedure which uses asymmetric key encryption and necessitates generation of at least one asymmetric key pair. Specifically, user's remote device negotiates with ASICS Digital servers the cipher suite and the key exchange algorithm that will be used for the handshake.



```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].
```

[43] *See, e.g.,* https://source.android.com/security/encryption/, Last accessed on Mar 19, 2020



```
DHE_RSA              RSA public key; the certificate MUST allow the
ECDHE_RSA            key to be used for signing (the
                     digitalSignature bit MUST be set if the key
                     usage extension is present) with the signature
                     scheme and hash algorithm that will be employed
                     in the server key exchange message.
                     Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS              DSA public key; the certificate MUST allow the
                     key to be used for signing with the hash
                     algorithm that will be employed in the server
                     key exchange message.

DH_DSS               Diffie-Hellman public key; the keyAgreement bit
DH_RSA               MUST be set if the key usage extension is
                     present.

ECDH_ECDSA           ECDH-capable public key; the public key MUST
ECDH_RSA             use a curve and point format supported by the
                     client, as described in [TLSECC].

ECDHE_ECDSA          ECDSA-capable public key; the certificate MUST
                     allow the key to be used for signing with the
                     hash algorithm that will be employed in the
                     server key exchange message.  The public key
                     MUST use a curve and point format supported by
                     the client, as described in  [TLSECC].
```

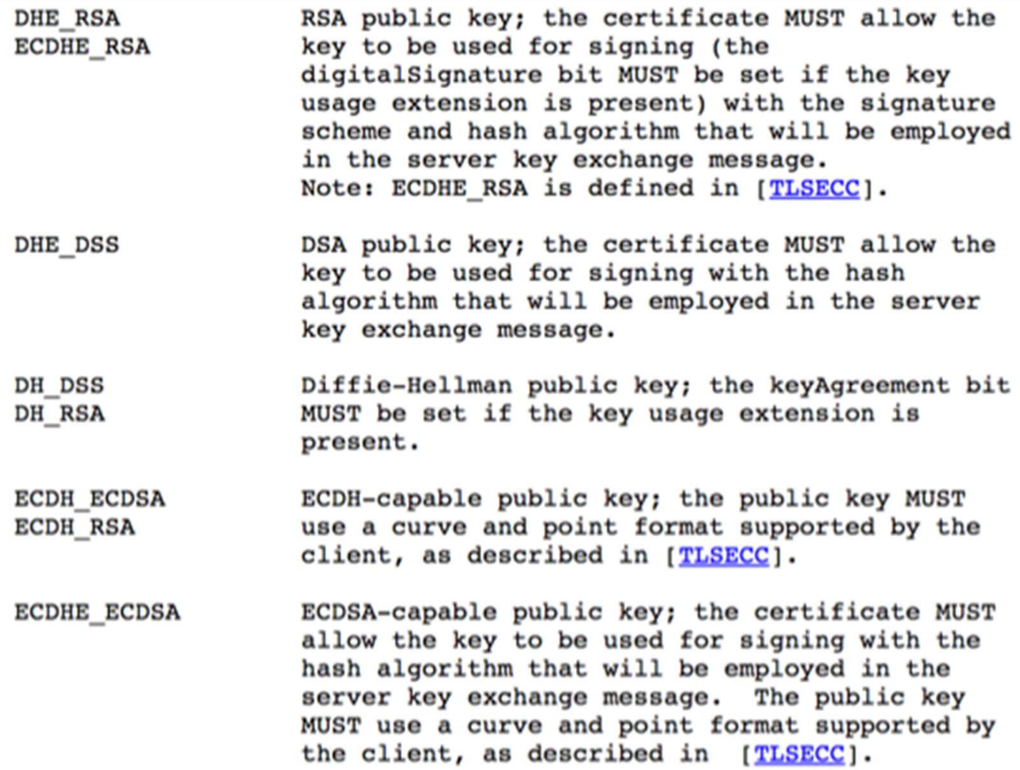
Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

Each of these algorithms necessitates generating one or more asymmetric key pairs – that are in turn used to compute a shared master secret for encrypting communication between ASICS Digital and user's remote device.

For **RSA and RSA _PSK**, ASICS Digital server generates an RSA public-private key pair.

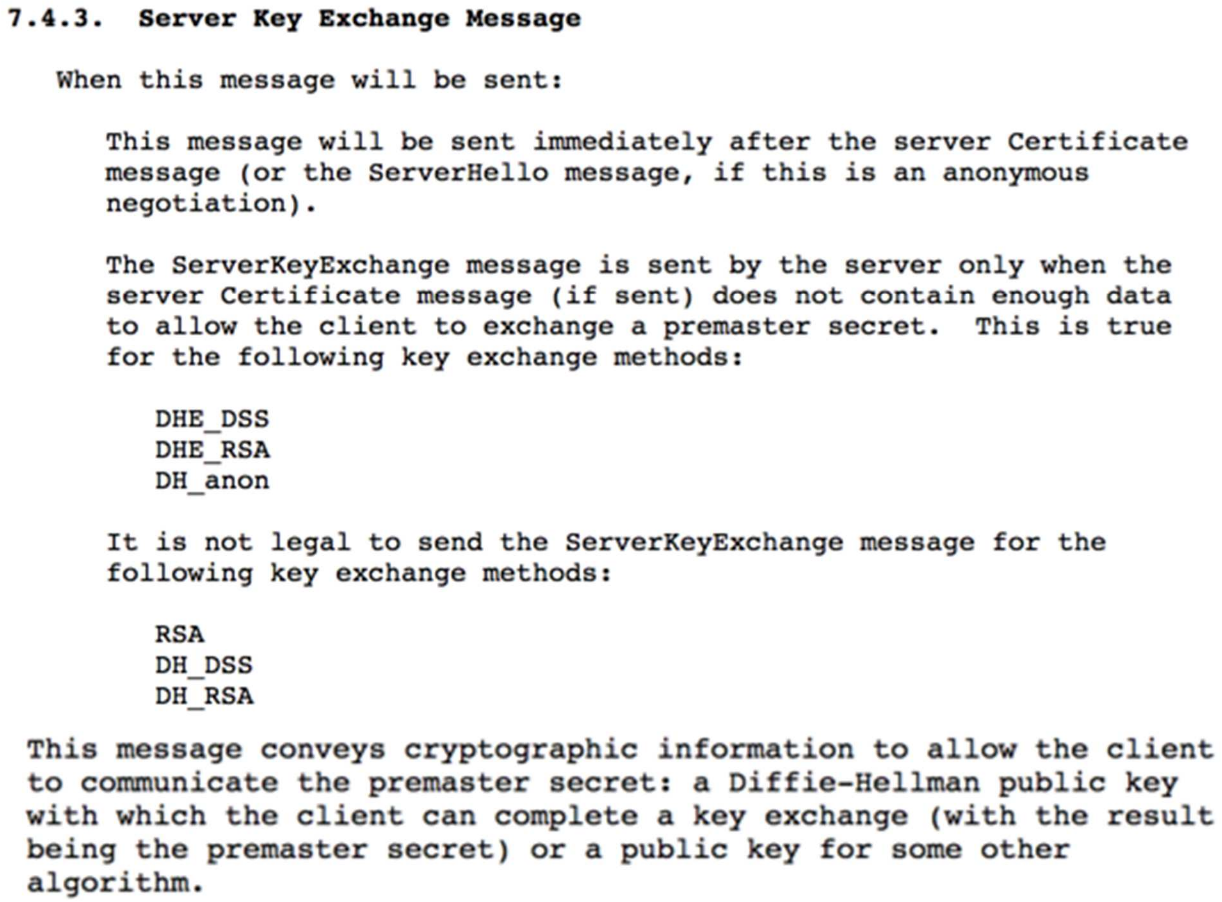
	For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, ASICS Digital server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[44]. The user's remote device also generates a second Diffie-Hellman public-private key pair. For **DHE_DSS and DH_DSS**, ASICS Digital server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. For **ECDH_ECDSA and ECDHE_ECDSA**, ASICS Digital server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair.

[44] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN 1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020



```
DHE_RSA              RSA public key; the certificate MUST allow the
ECDHE_RSA            key to be used for signing (the
                     digitalSignature bit MUST be set if the key
                     usage extension is present) with the signature
                     scheme and hash algorithm that will be employed
                     in the server key exchange message.
                     Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS              DSA public key; the certificate MUST allow the
                     key to be used for signing with the hash
                     algorithm that will be employed in the server
                     key exchange message.

DH_DSS               Diffie-Hellman public key; the keyAgreement bit
DH_RSA               MUST be set if the key usage extension is
                     present.

ECDH_ECDSA           ECDH-capable public key; the public key MUST
ECDH_RSA             use a curve and point format supported by the
                     client, as described in [TLSECC].

ECDHE_ECDSA          ECDSA-capable public key; the certificate MUST
                     allow the key to be used for signing with the
                     hash algorithm that will be employed in the
                     server key exchange message.  The public key
                     MUST use a curve and point format supported by
                     the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020



7.4.3. Server Key Exchange Message

When this message will be sent:

This message will be sent immediately after the server Certificate message (or the ServerHello message, if this is an anonymous negotiation).

The ServerKeyExchange message is sent by the server only when the server Certificate message (if sent) does not contain enough data to allow the client to exchange a premaster secret. This is true for the following key exchange methods:

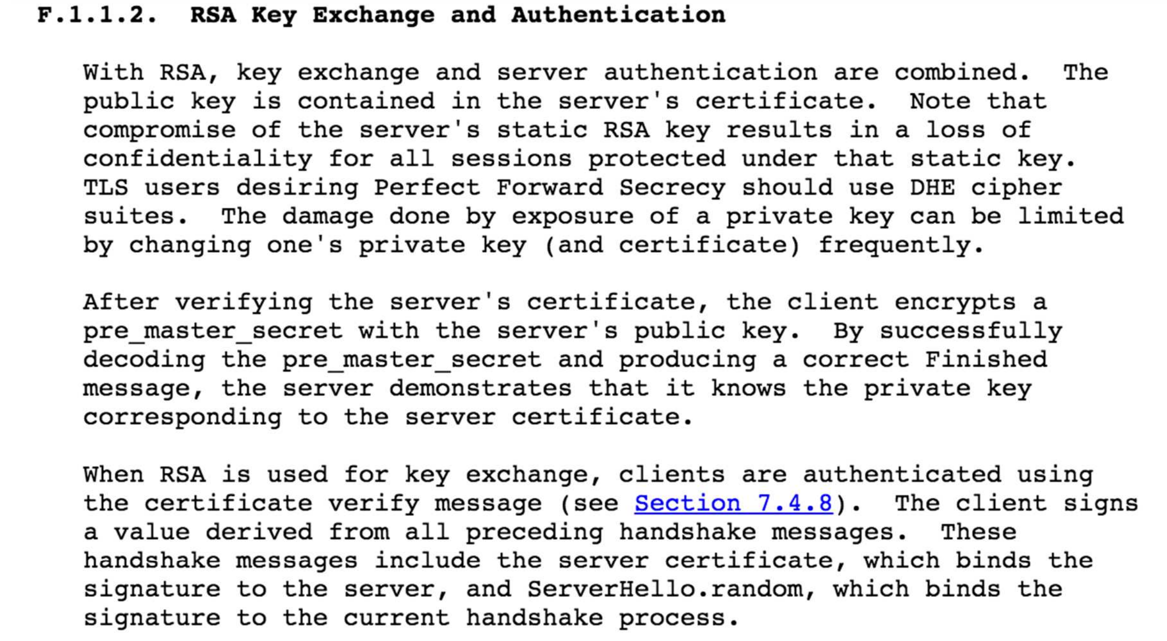
DHE_DSS
DHE_RSA
DH_anon

It is not legal to send the ServerKeyExchange message for the following key exchange methods:

RSA
DH_DSS
DH_RSA

This message conveys cryptographic information to allow the client to communicate the premaster secret: a Diffie-Hellman public key with which the client can complete a key exchange (with the result being the premaster secret) or a public key for some other algorithm.

Source: https://tools.ietf.org/html/rfc5246 at 50-51, Last accessed on Mar 19, 2020

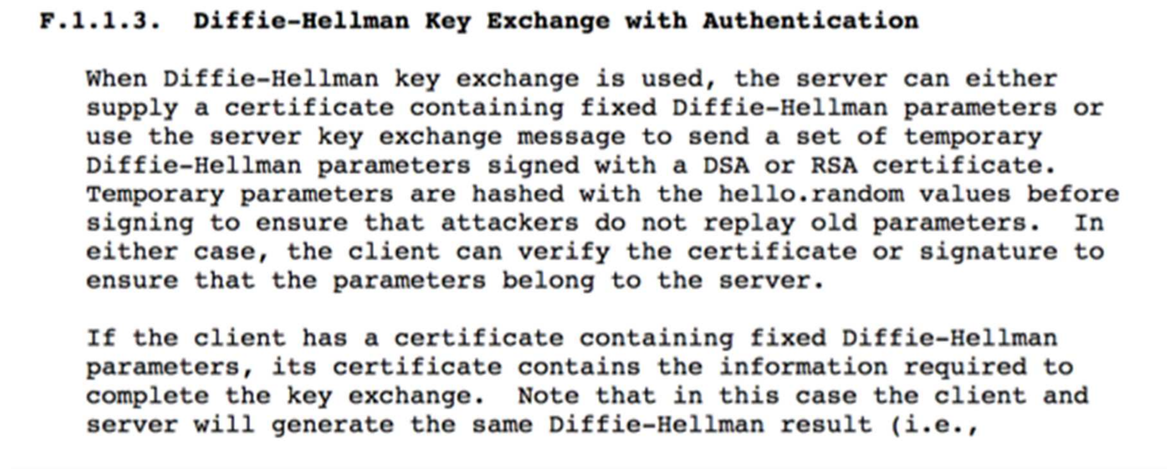


F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

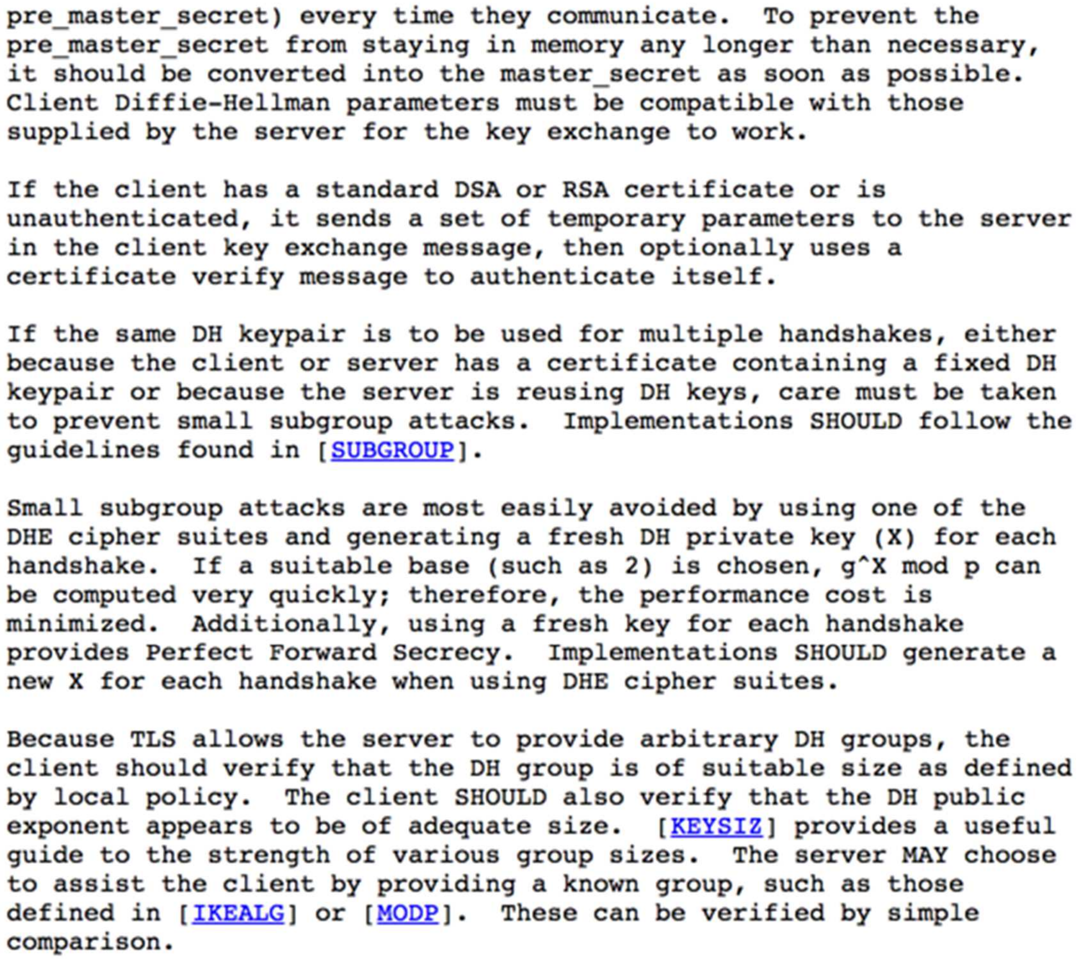
When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.



F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,



```
pre_master_secret) every time they communicate.  To prevent the
pre_master_secret from staying in memory any longer than necessary,
it should be converted into the master_secret as soon as possible.
Client Diffie-Hellman parameters must be compatible with those
supplied by the server for the key exchange to work.

If the client has a standard DSA or RSA certificate or is
unauthenticated, it sends a set of temporary parameters to the server
in the client key exchange message, then optionally uses a
certificate verify message to authenticate itself.

If the same DH keypair is to be used for multiple handshakes, either
because the client or server has a certificate containing a fixed DH
keypair or because the server is reusing DH keys, care must be taken
to prevent small subgroup attacks.  Implementations SHOULD follow the
guidelines found in [SUBGROUP].

Small subgroup attacks are most easily avoided by using one of the
DHE cipher suites and generating a fresh DH private key (X) for each
handshake.  If a suitable base (such as 2) is chosen, g^X mod p can
be computed very quickly; therefore, the performance cost is
minimized.  Additionally, using a fresh key for each handshake
provides Perfect Forward Secrecy.  Implementations SHOULD generate a
new X for each handshake when using DHE cipher suites.

Because TLS allows the server to provide arbitrary DH groups, the
client should verify that the DH group is of suitable size as defined
by local policy.  The client SHOULD also verify that the DH public
exponent appears to be of adequate size.  [KEYSIZ] provides a useful
guide to the strength of various group sizes.  The server MAY choose
to assist the client by providing a known group, such as those
defined in [IKEALG] or [MODP].  These can be verified by simple
comparison.
```

Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020

The generated asymmetric key pairs are then used to compute a shared master secret which is then used to encrypt subsequent communications between ASICS Digital and the user’s remote device so that they are resistant to observation during transit.

	For **RSA and RSA _PSK**, the RSA public-private key pair is used to encrypt a random premaster secret which is in turn used by ASICS Digital server and the user's remote device to compute a master secret. ASICS Digital and the user's remote device use the master secret for encrypting and decrypting communication messages. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, ASICS Digital server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[45]. The user's remote device also generates a second Diffie-Hellman public-private key pair. ASICS Digital server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Google's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret for encrypting and decrypting communication messages. For **DHE_DSS and DH_DSS**, ASICS Digital server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. ASICS Digital server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Google's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret for encrypting and decrypting communication messages. For **ECDH_ECDSA and ECDHE_ECDSA**, ASICS Digital server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. ASICS Digital server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Google's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret for encrypting and decrypting communication messages.

[45] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN
1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020